

**UNIVERSIDAD DE LAS FUERZAS ARMADAS  
ESPE**



**NOMBRES:** MIDEROS SAMIR, MIRANDA ALISON, MORÁN DAVID,  
VIVANCO GABRIEL

**NRC:** 27837

**FECHA:** 26/11/2025

Tema:

“Taller CRUD estudiantes Patrones de Diseño”

**Análisis y diseño del software**

**PERIODO 2025-2026**

## ANTES Y DESPUÉS - IMPLEMENTACIÓN DE PATRONES DE DISEÑO

### INTRODUCCIÓN

En este documento se presenta la evolución del proyecto CRUD de estudiantes, mostrando el estado antes y después de implementar dos patrones de diseño adicionales al patrón Singleton ya existente

### ESTADO INICIAL (ANTES)

#### Patrones Implementados:

- Singleton Pattern: Solo en las clases EstudianteRepository, EstudianteService y EstudianteUI

#### Estructura del Proyecto:

```
src/main/java/ec/edu/espe/  
├── datos/  
│   ├── model/  
│   │   └── Estudiante.java  
│   └── repository/  
│       └── EstudianteRepository.java  
├── logica_negocio/  
│   └── EstudianteService.java  
└── presentacion/  
    ├── EstudianteUI.java  
    └── Main.java
```

#### Características del Estado Inicial:

#### EstudianteRepository.java (ANTES):

- **Patrón Singleton:**
  - Una sola instancia del repositorio
  - Constructor privado
  - Método getInstance() para acceso global
- **Funcionalidades:**
  - CRUD básico (Create, Read, Update, Delete)
  - Almacenamiento en ArrayList
  - Validaciones básicas de ID duplicado
  - Métodos utilitarios (obtenerSiguieteld, contarEstudiantes, etc.)
- **Limitaciones:**
  - Búsquedas limitadas a búsqueda por ID únicamente
  - No hay sistema de notificaciones para cambios
  - No hay flexibilidad para diferentes tipos de búsqueda
  - No hay logging de operaciones
  - No hay estadísticas de uso

### ESTADO FINAL (DESPUÉS)

### Patrones Implementados:

1. **Singleton Pattern:** Mantenido en las mismas clases
2. **Strategy Pattern:** Añadido para diferentes estrategias de búsqueda
3. **Observer Pattern:** Añadido para notificar cambios en el repositorio

### Nueva Estructura del Proyecto:

```
src/main/java/ec/edu/espe/
├── datos/
│   ├── model/
│   │   └── Estudiante.java
│   └── repository/
│       ├── EstudianteRepository.java (MODIFICADO)
│       ├── strategy/ (NUEVO)
│       │   ├── IBusquedaStrategy.java
│       │   ├── BusquedaPorNombre.java
│       │   ├── BusquedaPorEdad.java
│       │   └── BusquedaPorId.java
│       └── observer/ (NUEVO)
│           ├── IRepositoryObserver.java
│           ├── LogObserver.java
│           └── EstadisticasObserver.java
├── logica_negocio/
│   └── EstudianteService.java
└── presentacion/
    ├── EstudianteUI.java
    └── Main.java
```

### PATRÓN STRATEGY (NUEVO)

#### Propósito:

Permite intercambiar algoritmos de búsqueda en tiempo de ejecución sin modificar el código cliente.

#### Implementación:

##### 1. Interfaz Strategy:

```
public interface IBusquedaStrategy {
    List<Estudiante> buscar(List<Estudiante> estudiantes, String criterio);
    String getNombreEstrategia();
}
```

##### 2. Estrategias Concretas:

###### BusquedaPorNombre:

- Busca estudiantes por nombre (coincidencia parcial, insensible a mayúsculas)

###### BusquedaPorEdad:

- Busca por edad exacta (ej: "20")
- Busca por rango de edad (ej: "18-25")

###### BusquedaPorId:

- Busca estudiante por ID específico

#### Nuevas Funcionalidades en EstudianteRepository:

```
// Cambio dinámico de estrategia
public void cambiarEstrategiaBusqueda(IBusquedaStrategy nuevaEstrategia)

// Búsqueda usando la estrategia actual
public List<Estudiante> buscarConEstrategia(String criterio)

// Métodos específicos para cada tipo de búsqueda
public List<Estudiante> buscarPorNombre(String nombre)
public List<Estudiante> buscarPorEdad(String edad)
```

#### Ventajas del Strategy Pattern:

1. **Flexibilidad:** Fácil agregar nuevos tipos de búsqueda
2. **Intercambiable:** Cambio de algoritmo en tiempo de ejecución
3. **Extensible:** Sin modificar código existente
4. **Reutilizable:** Estrategias pueden usarse en otros contextos

### PATRÓN OBSERVER (NUEVO)

#### Propósito:

Notifica automáticamente a múltiples objetos cuando ocurren cambios en el repositorio.

#### Implementación:

##### 1. Interfaz Observer:

```
public interface IRepositoryObserver {
    void onEstudianteAgregado(Estudiante estudiante);
    void onEstudianteEditado(Estudiante anterior, Estudiante nuevo);
    void onEstudianteEliminado(Estudiante estudiante);
    void onRepositorioLimpiado(int cantidadEliminados);
}
```

##### 2. Observadores Concretos:

###### LogObserver:

- Registra todas las operaciones con timestamp
- Muestra información detallada de cambios
- Útil para auditoría y debugging

###### EstadísticasObserver:

- Lleva contadores de operaciones realizadas
- Proporciona resumen de estadísticas
- Útil para métricas y análisis de uso

#### Nuevas Funcionalidades en EstudianteRepository:

```
// Gestión de observadores
public void agregarObservador(IRepositoryObserver observador)
public void removerObservador(IRepositoryObserver observador)

// Notificaciones automáticas (métodos privados)
private void notificarEstudianteAgregado(Estudiante estudiante)
private void notificarEstudianteEditado(Estudiante anterior, Estudiante nuevo)
private void notificarEstudianteEliminado(Estudiante estudiante)
private void notificarRepositorioLimpiado(int cantidadEliminados)
```

### Ventajas del Observer Pattern:

- **Desacoplamiento:** Los observadores no conocen detalles del repositorio
- **Extensibilidad:** Fácil agregar nuevos tipos de observadores
- **Notificación automática:** Sin llamadas manuales para informar cambios
- **Múltiples observadores:** Un cambio notifica a todos los interesados

### BENEFICIOS DE LA MEJORA

#### 1. Búsqueda Avanzada (Strategy Pattern):

- **Antes:** Solo búsqueda por ID
- **Después:** Búsqueda por nombre, edad, rango de edad, ID
- **Impacto:** Funcionalidad más rica para los usuarios

#### 2. Monitoreo y Logging (Observer Pattern):

- **Antes:** Sin logging de operaciones
- **Después:** Log automático con timestamps + estadísticas
- **Impacto:** Mejor debugging y análisis de uso

#### 3. Extensibilidad:

- **Antes:** Modificar código para nuevas funcionalidades
- **Después:** Agregar nuevas estrategias/observadores sin modificar código existente
- **Impacto:** Desarrollo más ágil y mantenible

#### 4. Arquitectura Mejorada:

- **Antes:** Repositorio monolítico
- **Después:** Responsabilidades distribuidas en patrones especializados
- **Impacto:** Código más limpio y modular

### COMPARACIÓN ENTRE PATRONES DE DISEÑO

#### EXTRACTOS DE CÓDIGO: ANTES vs DESPUÉS

- EstudianteRepository.java
- ANTES (Solo Singleton)

```
public class EstudianteRepository {

    private static EstudianteRepository instance;
    private List<Estudiante> estudiantes;
```

```

/**
 * Constructor privado para implementar Singleton
 */
private EstudianteRepository() {
    this.estudiantes = new ArrayList<>();
    inicializarDatosPrueba(); // Datos iniciales para pruebas
}

/**
 * Método para obtener la única instancia del repositorio
 */
public static EstudianteRepository getInstance() {
    if (instance == null) {
        instance = new EstudianteRepository();
    }
    return instance;
}

/**
 * Agregar un nuevo estudiante al repositorio
 * @param estudiante El estudiante a agregar
 * @return true si se agregó exitosamente, false si ya existe el ID
 */
public boolean agregar(Estudiante estudiante) {
    // Verificar que no exista un estudiante con el mismo ID
    if (buscarPorId(estudiante.getId()).isPresent()) {
        return false; // Ya existe un estudiante con ese ID
    }
    return estudiantes.add(estudiante);
}

/**
 * Editar un estudiante existente
 * @param estudiante El estudiante con los nuevos datos
 * @return true si se editó exitosamente, false si no se encontró
 */
public boolean editar(Estudiante estudiante) {
    for (int i = 0; i < estudiantes.size(); i++) {
        if (estudiantes.get(i).getId() == estudiante.getId()) {
            estudiantes.set(i, estudiante);
            return true;
        }
    }
}

```

```

        return false; // No se encontró el estudiante
    }

    /**
     * Eliminar un estudiante por su ID
     * @param id El ID del estudiante a eliminar
     * @return true si se eliminó exitosamente, false si no se encontró
     */
    public boolean eliminar(int id) {
        return estudiantes.removeIf(estudiante -> estudiante.getId() ==
id);
    }

```

## DESPUÉS (Singleton + Strategy + Observer)

```

public class EstudianteRepository {
    // SINGLETON (Mantenido)
    private static EstudianteRepository instance;
    private List<Estudiante> estudiantes;

    // NUEVOS COMPONENTES PARA PATRONES
    private IBusquedaStrategy estrategiaBusqueda;    // Strategy Pattern
    private List<IRepositoryObserver> observadores; // Observer Pattern

    private EstudianteRepository() {
        this.estudiantes = new ArrayList<>();
        this.observadores = new ArrayList<>();
        this.estrategiaBusqueda = new BusquedaPorNombre(); // Estrategia por defecto

        // AUTO-REGISTRO DE OBSERVADORES
        agregarObservador(new LogObserver());
        agregarObservador(new EstadisticasObserver());

        inicializarDatosPrueba();
    }

    // BÚSQUEDAS FLEXIBLES CON STRATEGY
    public List<Estudiante> buscarConEstrategia(String criterio) {
        return estrategiaBusqueda.buscar(estudiantes, criterio);
    }

    public List<Estudiante> buscarPorNombre(String nombre) {
        IBusquedaStrategy estrategiaAnterior = this.estrategiaBusqueda;

```

```

    BusquedaStrategy estrategiaAnterior = this.estrategiaBusqueda;
    this.estrategiaBusqueda = new BusquedaPorNombre();
    List<Estudiante> resultado = buscarConEstrategia(nombre);
    this.estrategiaBusqueda = estrategiaAnterior;
    return resultado;
}

public List<Estudiante> buscarPorEdad(String edad) {
    IBusquedaStrategy estrategiaAnterior = this.estrategiaBusqueda;
    this.estrategiaBusqueda = new BusquedaPorEdad();
    List<Estudiante> resultado = buscarConEstrategia(edad);
    this.estrategiaBusqueda = estrategiaAnterior;
    return resultado;
}

// OPERACIONES CON NOTIFICACIONES AUTOMÁTICAS
public boolean agregar(Estudiante estudiante) {
    if (buscarPorId(estudiante.getId()).isPresent()) {
        return false;
    }

    boolean resultado = estudiantes.add(estudiante);

    // NOTIFICACIÓN AUTOMÁTICA
    if (resultado) {
        notificarEstudianteAgregado(estudiante);
    }

    return resultado;
}

```

Activar Windows  
Ve a Configuración para

## COMPARACIÓN DE CAPACIDADES BÚSQUEDAS (ANTES)

```

// Solo búsqueda por ID
Optional<Estudiante> estudiante = repo.buscarPorId(1);

// Para buscar por nombre: IMPOSIBLE sin modificar código
// Para buscar por edad: IMPOSIBLE sin modificar código
// Para buscar por rango: IMPOSIBLE sin modificar código

```

## DESPUÉS

```
// Búsquedas flexibles y potentes
List<Estudiante> porNombre = repo.buscarPorNombre("juan");
List<Estudiante> porEdad = repo.buscarPorEdad("20");
List<Estudiante> porRango = repo.buscarPorEdad("18-25");

// Cambio dinámico de algoritmo
repo.cambiarEstrategiaBusqueda(new BusquedaPorId());
List<Estudiante> resultado = repo.buscarConEstrategia("1");
```

## TABLA COMPARATIVA DE PATRONES DE DISEÑO

Aspecto	Singleton	Strategy	Observer
<b>Propósito Principal</b>	Garantizar que exista una única instancia en todo el sistema.	Permitir intercambiar algoritmos de manera flexible y dinámica.	Notificar automáticamente a múltiples componentes cuando ocurre un cambio.
<b>Problema que Resuelve</b>	Evita la creación de múltiples instancias no deseadas.	Elimina algoritmos fijos y difíciles de modificar.	Reduce el código acoplado para manejar notificaciones.
<b>Beneficio en el Contexto</b>	Asegura consistencia de datos al centralizar el acceso.	Facilita búsquedas o procesos configurables.	Permite generar registros (logs) y estadísticas sin modificar la lógica principal.
<b>Ejemplo Práctico</b>	Uso de <code>EstudianteRepository.getInstance()</code> .	Cambiar de un método de búsqueda por nombre a uno por edad.	Registrar automáticamente eventos al agregar un estudiante.
<b>Extensibilidad</b>	Media.	Muy alta.	Muy alta.
<b>Complejidad de Implementación</b>	Baja (simple).	Media.	Media a alta.
<b>Mantenibilidad</b>	Buena.	Excelente.	Muy buena.

<b>Testabilidad</b>	Limitada debido a la única instancia.	Excelente.	Muy buena.
<b>Acoplamiento</b>	Bajo.	Muy bajo.	Bajo.
<b>Reutilización</b>	Limitada.	Excelente.	Muy buena.

## ¿POR QUÉ SOLO SINGLETON NO ERA SUFICIENTE?

### Limitaciones del Sistema Original (versión corta)

El sistema antes sólo permitía buscar por ID, lo que hacía que agregar nuevas búsquedas (como por nombre o edad) obligará a modificar el repositorio y recompilar todo.

Además, no existía ningún registro de lo que pasaba: cuando se agregaba o eliminaba un estudiante no quedaba evidencia, lo que hacía imposible hacer auditorías o rastrear errores.

En general, el sistema tenía funciones limitadas, sin logs, sin estadísticas y con código muy rígido.

```
Optional<Estudiante> estudiante = repo.buscarPorId(1);
```

### Ventajas del Sistema Mejorado (versión corta)

Con Strategy, ahora el sistema permite muchas formas de búsqueda sin tocar el repositorio.

Agregar una nueva búsqueda solo requiere crear una clase nueva.

Con Observer, cada operación queda registrada automáticamente con logs y estadísticas.

El sistema ahora es flexible, extensible y mucho más fácil de mantener.

```
// Múltiples formas de buscar sin cambiar código base
```

```
List<Estudiante> porNombre = repo.buscarPorNombre("juan");
```

```
List<Estudiante> jovenes = repo.buscarPorEdad("18-25");
```

## ¿POR QUÉ ELEGÍ STRATEGY Y OBSERVER? (Versión Corta y Clara)

### Strategy Pattern – Justificación

#### Problema:

El sistema sólo permitía buscar por ID, lo cual es muy limitado para un CRUD real.

#### Por qué Strategy encaja perfecto:

Permite tener múltiples tipos de búsqueda sin modificar el repositorio.

Ejemplos reales que ahora son posibles:

- Buscar por nombre: "juan" → "Juan Pérez"
- Buscar por edad exacta: "20"
- Buscar por rango de edad: "18-25"

## Observer Pattern – Justificación

### Problema:

Antes no había logs ni estadísticas, así que no se podía auditar, depurar ni medir el uso del sistema.

### Por qué Observer es ideal:

Permite monitorear automáticamente cada operación sin tocar el CRUD.

- `repo.agregar(estudiante);` // Se registra el log y la estadística automáticamente

Y permite varios observadores actuando en paralelo:

- `LogObserver` → para auditoría
- `EstadisticasObserver` → para métricas

## IMPACTO EN EL CONTEXTO DEL PROGRAMA

La incorporación de Strategy y Observer transformó el sistema en algo más flexible, útil y fácil de mantener. Para el usuario final, el cambio es evidente: antes solo podía ver la lista completa de estudiantes; ahora puede realizar búsquedas específicas según lo que necesite, como estudiantes jóvenes entre 18 y 20 años:

```
List<Estudiante> jovenes = repo.buscarPorEdad("18-20");
```

Para el desarrollador, el sistema dejó de ser rígido. Antes, agregar una nueva funcionalidad exigía modificar el repositorio y arriesgar algo. Ahora basta con implementar una nueva estrategia sin tocar el código existente:

```
public class BusquedaPorPromedio implements IBusquedaStrategy { ... }
```

También el administrador del sistema obtiene beneficios. Antes no existían registros de uso ni evidencia de errores, pero ahora el sistema genera logs y estadísticas automáticamente:

```
? Editando estudiante demo...
[LOG] 26/11/2025 19:48:10 - EDITADO:
    Anterior: Estudiante{id=99, nombres='Estudiante Demo', edad=25}
    Nuevo: Estudiante{id=99, nombres='Estudiante Demo Editado', edad=26}
[ESTADÍSTICAS] Estudiantes editados: 1
```

## CONCLUSIONES

La implementación de los patrones Strategy y Observer en el repositorio de estudiantes ha transformado significativamente las capacidades del sistema:

**Logros Alcanzados:**

- Funcionalidad expandida sin romper código existente
- Arquitectura más flexible y extensible
- Mejor monitoreo y logging automático
- Búsquedas avanzadas para una mejor experiencia de usuario

**Calidad del Código:**

- **Mantenibilidad:** Fácil agregar nuevas funcionalidades
- **Testabilidad:** Componentes más pequeños y especializados
- **Reutilización:** Estrategias y observadores reutilizables
- **Separación de responsabilidades:** Cada patrón tiene su propósito específico

**Impacto en el Usuario Final:**

- **Búsquedas más potentes:** Por nombre, edad, rangos
- **Mejor debugging:** Logs detallados de todas las operaciones
- **Estadísticas de uso:** Para analizar patrones de utilización
- **Sistema más robusto:** Con notificaciones automáticas de cambios

En conclusión, el sistema pasó de ser básico a ser extensible y profesional. Se logró ampliar la funcionalidad, implementar monitoreo real, mejorar la organización interna del código y hacerlo crecer sin romper lo que ya funcionaba. El usuario obtiene una experiencia más útil, el desarrollador trabaja con un sistema más ordenado y el administrador puede monitorear todo en tiempo real. Con esta base sólida, resulta sencillo añadir nuevas estrategias, observadores más avanzados, almacenamiento en base de datos o incluso exponer todo como una API REST.