Universidad de las Fuerzas Armadas ESPE

Departamento: Ciencias de la Computación

Carrera: Ingeniería de Software

---

1. Información General

- **Asignatura:** Análisis y Diseño de Software
- **Apellidos y nombres de los estudiantes:**
  - Chavez Kleber
  - Guacan Alexander
  - Trejo Alex
- **NRC:** 23305
- **Fecha de realización: 09/07/2025**

---

2. Resumen Ejecutivo

Este informe analiza la aplicación práctica de los principios de diseño de software SOLID en el desarrollo de un sistema de gestión de estudiantes, construido sobre una arquitectura de 3 capas y el patrón Modelo-Vista-Controlador (MVC) en Java. El objetivo principal es demostrar cómo la adhesión a estos principios transforma un código funcional en un sistema robusto, mantenible y extensible.

La implementación del proyecto demostró que la aplicación rigurosa de SOLID es una inversión estratégica que produce beneficios tangibles. Se concluyó que los principios SOLID conducen a un sistema con un bajo acoplamiento y una alta cohesión, lo que se traduce directamente en:

- Mayor Mantenibilidad: Los cambios en un área del sistema tienen un impacto mínimo en otras, reduciendo el riesgo de introducir errores.
- Flexibilidad Superior: El sistema está preparado para la extensión. Agregar nuevas funcionalidades o cambiar componentes (como el método de persistencia de datos) se puede lograr sin modificar el código existente.
- Testabilidad Mejorada: La separación de responsabilidades y la inversión de dependencias facilitan enormemente las pruebas unitarias aisladas de cada componente.

La recomendación principal de este informe es la adopción de los principios SOLID como pilar o herramienta fundamental en todos los ciclos de desarrollo de software. Esto incluye la integración en las revisiones de código, la capacitación continua del

equipo y la elección de arquitecturas que, por naturaleza, promuevan su aplicación. Invertir en un diseño SOLID desde el inicio reduce drásticamente la deuda técnica y el costo total de propiedad del software a largo plazo.

3. Introducción

En el desarrollo de software moderno, la creación de código que funcione es solo el primer paso. El verdadero desafío reside en construir sistemas que puedan evolucionar, adaptarse a nuevos requisitos y ser mantenidos por diferentes equipos a lo largo del tiempo. Los principios SOLID, acuñados por Robert C. Martin, son un conjunto de cinco directrices de diseño para la programación orientada a objetos que buscan abordar precisamente estos desafíos.

Este documento detalla la implementación de cada uno de estos cinco principios en el contexto de una aplicación Java para gestionar estudiantes. Analizaremos cómo cada principio fue aplicado en el código y el impacto directo que tuvo en la calidad general de la arquitectura.

4. Aplicación de los Principios SOLID

A continuación, se desglosa cada principio con ejemplos concretos extraídos del código del proyecto.

S – Principio de Responsabilidad Única (Single Responsibility Principle - SRP)

Definición: "Una clase debe tener una, y solo una, razón para cambiar". Esto significa que cada clase debe tener una única responsabilidad o propósito bien definido dentro del sistema.

Aplicación en el Proyecto: El SRP es la base de nuestra arquitectura de 3 capas. Cada clase tiene un propósito claramente delimitado:

1. **EstudianteUI (Vista):** Su única responsabilidad es la interacción con el usuario (mostrar menús, capturar datos). No sabe nada de lógica de negocio o cómo se almacenan los datos.
2. **EstudianteService (Controlador/Lógica)**: Su única responsabilidad es orquestar la lógica de negocio (validar datos, coordinar operaciones). No interactúa directamente con el usuario ni con la base de datos.
3. **EstudianteRepositoryImpl (DAO):** Su única responsabilidad es la persistencia de datos (agregar, leer, actualizar y eliminar estudiantes de una ArrayList).

Impacto y Beneficios:

Esta estricta separación de responsabilidades evita la creación de "clases dios" monolíticas. Si necesitamos cambiar la interfaz de usuario de consola a una gráfica, solo modificamos la capa de presentación. Si las reglas de negocio cambian, solo tocamos la capa de servicio. Esto hace que el código sea más fácil de entender, mantener y probar.

O – Principio Abierto/Cerrado (Open/Closed Principle - OCP)

Definición: "Las entidades de software (clases, módulos, funciones) deben estar abiertas para la extensión, pero cerradas para la modificación".

Aplicación en el Proyecto: Este principio se materializa a través del uso de la interfaz IEstudianteRepository. La clase EstudianteService no depende de la implementación concreta (EstudianteRepositoryImpl), sino de la abstracción.

```
// logica_negocio/EstudianteService.java

public class EstudianteService {

    // Depende de la ABSTRACCIÓN, no de una clase concreta

    private final IEstudianteRepository estudianteRepository;


    public EstudianteService(IEstudianteRepository estudianteRepository) {

        this.estudianteRepository = estudianteRepository;

    }


    public List<Estudiante> obtenerTodosLosEstudiantes() {

        // Usa la interfaz, sin saber qué clase la implementa

        return estudianteRepository.obtenerTodos();

    }

}
```

Impacto y Beneficios: El sistema está abierto a la extensión. Si mañana decidimos almacenar los datos en una base de datos PostgreSQL, podemos crear una nueva clase EstudianteRepositoryPostgresImpl que implemente IEstudianteRepository. Luego, en el Main, simplemente inyectamos la nueva instancia. El EstudianteService y la EstudianteUI no requieren ninguna modificación. Esto es fundamental para la escalabilidad y flexibilidad del software.

L – Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP)

Definición: "Los subtipos deben ser sustituibles por sus tipos base sin alterar la corrección del programa". En términos prácticos, si una clase B es un subtipo de A, deberíamos poder usar un objeto de B en cualquier lugar donde se espere un objeto de A.

Aplicación en el Proyecto: Nuestra aplicación respeta el LSP. El EstudianteService espera un objeto de tipo IEstudianteRepository. Como EstudianteRepositoryImpl implementa correctamente todos los métodos definidos en la interfaz sin cambiar su comportamiento esperado, es perfectamente sustituible.

```
// presentacion/Main.java

public class Main {

    public static void main(String[] args) {

        // Creamos una instancia del subtipo

        IEstudianteRepository repository = new EstudianteRepositoryImpl();



        // La pasamos a un método que espera el tipo base.

        // El programa funcionará correctamente porque el subtipo es sustituible.

        EstudianteService service = new EstudianteService(repository);



            }

}
```

Impacto y Beneficios: El LSP garantiza que nuestras abstracciones (interfaces) sean fiables. Nos da la confianza de que cualquier implementación futura de una interfaz funcionará correctamente con el resto del sistema, siempre que cumpla con el "contrato" de la interfaz. Esto previene errores sutiles y comportamientos inesperados.

I – Principio de Segregación de Interfaces (Interface Segregation Principle - ISP)

Definición: "Ningún cliente debe ser forzado a depender de métodos que no utiliza". Es preferible tener muchas interfaces pequeñas y específicas que una grande y monolítica.

Aplicación en el Proyecto: La interfaz IEstudianteRepository es un buen ejemplo de una interfaz cohesiva y específica. Contiene solo los métodos relacionados con las operaciones CRUD de estudiantes.

```
// datos/repository/IEstudianteRepository.java

public interface IEstudianteRepository {

    void agregar(Estudiante estudiante);

    List<Estudiante> obtenerTodos();

    Estudiante buscarPorId(int id);

    void actualizar(Estudiante estudianteActualizado);

    void eliminar(int id);

}
```

Si el sistema requiriera una funcionalidad no relacionada, como generar reportes, el ISP nos dice que no deberíamos agregar un método generarReporteCSV() a esta interfaz. En su lugar, crearíamos una nueva interfaz, como IReporteRepository, para no obligar a las clases de persistencia simples a implementar métodos de reporte que no necesitan.

Impacto y Beneficios: El ISP previene el "engorde" de las interfaces y mantiene las clases limpias y enfocadas. Evita que las clases implementadoras tengan que proporcionar implementaciones vacías o lanzar excepciones para métodos que no les conciernen, lo que resulta en un código más limpio y menos propenso a errores.

D – Principio de Inversión de Dependencias (Dependency Inversion Principle - DIP)

Definición: "a) Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones. b) Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones".

Aplicación en el Proyecto: Este es quizás el principio más influyente en nuestra arquitectura. El módulo de alto nivel (EstudianteService) no depende directamente del módulo de bajo nivel (EstudianteRepositoryImpl). En cambio, ambos dependen de la abstracción IEstudianteRepository. La dependencia es "invertida" porque el control de la creación de instancias se externaliza, generalmente a través de la Inyección de Dependencias.

```
// logica_negocio/EstudianteService.java

public class EstudianteService {

    // La clase no crea su propia dependencia. La RECIBE.

    private final IEstudianteRepository estudianteRepository;



    // Inyección de dependencias a través del constructor

    public EstudianteService(IEstudianteRepository estudianteRepository) {

        this.estudianteRepository = estudianteRepository;

    }

    // ...

}
```

Impacto y Beneficios: El DIP es la clave para un sistema desacoplado. Permite que los componentes se desarrollen y prueben de forma independiente. Para probar EstudianteService, podemos "inyectarle" una implementación falsa (un mock) de IEstudianteRepository que simule el acceso a datos, sin necesidad de una base de datos real. Esto acelera drásticamente el ciclo de desarrollo y pruebas.

5. Conclusiones

Como conclusión, la adopción de los fundamentos SOLID durante el desarrollo del sistema de gestión de estudiantes demuestra un impacto transformador en el control y manejo de la aplicación. Esta metodología fomenta una notable independencia entre los módulos, principalmente a través del Principio de Responsabilidad Única (SRP) y el Principio de Inversión de Dependencias (DIP). Al aislar las responsabilidades en capas distintas —Presentación, Lógica de Negocio y Acceso a Datos—, se logra que las modificaciones en un componente, como cambiar la interfaz de usuario, no generen efectos colaterales imprevistos en la lógica subyacente.

Asimismo, se establece un control riguroso sobre la evolución del software; el Principio Abierto/Cerrado (OCP), facilitado por el uso de interfaces como IEstudianteRepository, asegura que el sistema pueda extenderse para soportar nuevas funcionalidades —por ejemplo, una nueva base de datos— sin necesidad de alterar el código base existente y ya probado. En conjunto, estos principios no solo mejoran la estructura del código, sino

que reducen la deuda técnica y convierten el software en un activo estratégico, fácil de mantener, probar y escalar a lo largo de su ciclo de vida.

6. Recomendaciones

Se recomienda la formalización de la aplicación de los principios SOLID como una política estándar en todos los ciclos de desarrollo de software. Esto implica su integración obligatoria en los estándares de codificación y como criterio fundamental durante las revisiones de código (code reviews), asegurando que cada nueva pieza de software contribuya a la salud general de la arquitectura en lugar de degradarla.

Adicionalmente, es crucial que desde la fase de concepción de un proyecto se elijan patrones arquitectónicos que por su naturaleza promuevan un diseño desacoplado, como la arquitectura de 3 capas, la arquitectura hexagonal o la Clean Architecture, pues estos actúan como un andamiaje que facilita la implementación natural de principios como la Inversión de Dependencias. Finalmente, para garantizar la sostenibilidad de estas prácticas, se debe invertir de forma continua en la capacitación del equipo de desarrollo y asignar tiempo específico para la refactorización de código existente