Aplicación de los Principios SOLID en el Sistema de Gestión de Estudiantes

Grupo 6
Moreira Erick
Granda Carlos
Coloma Kevin

10 de julio de 2025

1. Introducción

Los principios SOLID son un conjunto de buenas prácticas en la programación orientada a objetos que ayudan a diseñar sistemas mantenibles, escalables y comprensibles. En este documento se describe cómo se aplicaron estos principios en el desarrollo de un sistema de gestión de estudiantes.

1. ¿Qué es SRP?

El **Principio de Responsabilidad Única (Single Responsibility Principle - SRP)** es el primero del conjunto de principios **SOLID**, propuestos por Robert C. Martin.

SRP establece que una clase debe tener *una única razón para cambiar*, es decir, debe estar enfocada en una sola responsabilidad o funcionalidad dentro del sistema.

Este principio fomenta la cohesión, facilita el mantenimiento y mejora la escalabilidad del software.

> *Una clase debería tener una, y solo una, razón para cambiar.*
> (Robert C. Martin, 2003)

2. Ejemplo que NO cumple SRP

A continuación, se presenta una clase que infringe el principio SRP al mezclar responsabilidades:

```
public class Estudiante {
    private int id;
    private String nombre;

    public Estudiante(int id, String nombre) {
```

```
        this.id = id;
        this.nombre = nombre;
    }

    public void guardarEnBD() {
        System.out.println("Guardando en BD...");
    }

    public void imprimir() {
        System.out.println("Estudiante: " + nombre);
    }
}
```

Responsabilidades mezcladas:

- Modelo de datos (atributos `id` y `nombre`)
- Persistencia (guardar en base de datos)
- Presentación (mostrar en consola)

3. Análisis del Problema

Combinar múltiples funciones en una sola clase implica:

- **Alto acoplamiento:** Las funcionalidades están fuertemente ligadas.
- **Baja cohesión:** La clase no se especializa en una tarea concreta.
- **Dificultad para mantenimiento y pruebas.**
- **Escalabilidad limitada:** Cambios menores pueden romper todo el sistema.

4. Ejemplo que SÍ cumple SRP

Dividimos las responsabilidades en clases especializadas:

A. Clase Modelo (Entidad)

```
public class Estudiante {
    private int id;
    private String nombre;

    public Estudiante(int id, String nombre) {
        this.id = id;
        this.nombre = nombre;
    }
```

```
    public int getId() { return id; }
    public String getNombre() { return nombre; }
}
```

B. Clase DAO (Persistencia)

```
public class EstudianteDAO {
    public void guardar(Estudiante e) {
        System.out.println("Guardando␣estudiante␣en␣la␣BD...");
    }
}
```

C. Clase Vista (Presentación)

```
public class EstudiantePrinter {
    public void imprimir(Estudiante e) {
        System.out.println("Estudiante:␣" + e.getNombre());
    }
}
```

5. Ventajas de aplicar SRP

- **Mantenibilidad:** Se modifica solo la clase afectada.
- **Reutilización:** Las clases especializadas pueden usarse en otros proyectos.
- **Facilidad para testeo unitario:** Cada clase se prueba de forma independiente.
- **Escalabilidad:** Se facilita la extensión del sistema sin afectar otras partes.

6. SRP en el ejercicio de Arquitectura MVC CRUD Estudiante

Descripción del problema

Este ejemplo consiste en una aplicación CRUD de estudiantes. Se divide en varias clases, cada una con una única responsabilidad, cumpliendo con el principio SRP. La lógica de presentación (UI), negocio (controlador), acceso a datos (DAO), y modelo (entidad) están separadas.

Análisis del Principio SRP

Cada clase cumple con una única razón para cambiar:

- **Clase** `Estudiante`: representa el modelo de datos.
- **Clase** `EstudianteDAO`: se encarga del almacenamiento y gestión de datos.
- **Clase** `EstudianteController`: contiene la lógica de negocio.
- **Clase** `MainGUI`: se encarga exclusivamente de la interacción con el usuario.

Código de ejemplo

Clase Modelo (`Estudiante`)

Listing 1: Clase Estudiante

```
package model;
public class Estudiante {
    private int id;
    private String apellidos;
    private String nombres;
    private int edad;

    public Estudiante(int id, String apellidos, String nombres, int edad)
       {
        this.id = id;
        this.apellidos = apellidos;
        this.nombres = nombres;
        this.edad = edad;
    }
    // Getters y Setters...
}
```

Clase DAO (`EstudianteDAO`)

Listing 2: Clase EstudianteDAO

```
package dao;
import java.util.*;
import model.Estudiante;

public class EstudianteDAO {
    private List<Estudiante> estudiantes = new ArrayList<>();

    public void agregar(Estudiante e) { estudiantes.add(e); }
    public List<Estudiante> listar() { return estudiantes; }
    public Estudiante buscarPorId(int id) {
        for (Estudiante e : estudiantes) {
            if (e.getId() == id) return e;
        }
```

```
        return null;
    }
    public boolean editar(Estudiante nuevo) {
        for (int i = 0; i < estudiantes.size(); i++) {
            if (estudiantes.get(i).getId() == nuevo.getId()) {
                estudiantes.set(i, nuevo);
                return true;
            }
        }
        return false;
    }
    public boolean eliminar(int id) {
        return estudiantes.removeIf(e -> e.getId() == id);
    }
}
```

Clase Controlador (`EstudianteController`)

Listing 3: Clase EstudianteController

```
package controller;
import dao.EstudianteDAO;
import model.Estudiante;
import java.util.List;

public class EstudianteController {
    private EstudianteDAO dao = new EstudianteDAO();

    public void crearEstudiante(int id, String apellidos, String nombres,
       int edad) {
        Estudiante e = new Estudiante(id, apellidos, nombres, edad);
        dao.agregar(e);
    }
    public List<Estudiante> obtenerTodos() { return dao.listar(); }
    public Estudiante buscarEstudiante(int id) { return dao.buscarPorId(id
       ); }
    public boolean actualizarEstudiante(int id, String apellidos, String
       nombres, int edad) {
        Estudiante actualizado = new Estudiante(id, apellidos, nombres,
           edad);
        return dao.editar(actualizado);
    }
    public boolean eliminarEstudiante(int id) { return dao.eliminar(id); }
}
```

Interfaz Gráfica (`MainGUI`)

Listing 4: Clase MainGUI - Interfaz gráfica (fragmento)

```
package ui;
import controller.EstudianteController;
import model.Estudiante;
```

```
import javax.swing.*;

public class MainGUI extends JFrame {
    private EstudianteController controller = new EstudianteController();
    private JTextField txtId = new JTextField(5);
    private JTextField txtApellidos = new JTextField(10);
    private JTextField txtNombres = new JTextField(10);
    private JTextField txtEdad = new JTextField(3);
    // Resto de componentes, eventos y m todos...
}
```

Justificación SRP

Este diseño facilita el mantenimiento y la escalabilidad del sistema. Por ejemplo:

- Si cambia la forma de almacenar los datos (persistencia), solo se modifica `EstudianteDAO`.
- Si cambia la lógica de negocio, se edita únicamente `EstudianteController`.
- Si se actualiza la interfaz gráfica, sólo se modifica `MainGUI`.

2. Principios SOLID Aplicados

2.1. Ejemplo aplicado del Principio SRP (Responsabilidad Única)

2.1.1. Descripción

Este principio establece que una clase debe tener una única razón para cambiar, es decir, debe tener una única responsabilidad.

2.1.2. Aplicación en el sistema

Se crearon clases específicas para separar responsabilidades:

- **Estudiante**: modelo con los datos del estudiante.
- **EstudianteDAO**: maneja el almacenamiento de los datos.
- **EstudianteController**: controla la lógica entre la vista y el modelo.

2.1.3. Ventajas obtenidas

- Fácil mantenimiento y comprensión.
- Separación clara de funciones.

2.2. Ejemplo aplicado del Principio OCP (Abierto/Cerrado)

2.2.1. Descripción

Una clase debe estar abierta a extensión pero cerrada a modificación.

2.2.2. Aplicación en el sistema

Se utilizó la interfaz `IEstudianteDAO` para permitir múltiples implementaciones del acceso a datos.

- Se pueden crear nuevas clases como `EstudianteArchivoDAO` sin modificar el controlador.

2.2.3. Ventajas obtenidas

- Menor riesgo de errores al modificar.
- Sistema fácilmente extensible.

2.3. Ejemplo aplicado del Principio LSP (Sustitución de Liskov)

2.3.1. Descripción

Las subclases deben poder sustituir a sus clases base sin alterar el comportamiento del programa.

2.3.2. Aplicación en el sistema

El controlador usa la interfaz `IEstudianteDAO`, lo que permite sustituir libremente sus implementaciones.

- `EstudianteDAO` y otras implementaciones pueden intercambiarse sin errores.

2.3.3. Ventajas obtenidas

- Mayor flexibilidad y reutilización del código.

2.4. Ejemplo aplicado del Principio ISP (Segregación de Interfaces)

2.4.1. Descripción

Las interfaces deben ser específicas, no obligando a implementar métodos innecesarios.

2.4.2. Aplicación en el sistema

La interfaz `IEstudianteDAO` contiene sólo métodos relevantes para la gestión de estudiantes.

- Las clases DAO solo implementan lo necesario.

2.4.3. Ventajas obtenidas

- Código más limpio y claro.
- Menor complejidad en las clases.

2.5. Ejemplo aplicado del Principio DIP (Inversión de Dependencias)

2.5.1. Descripción

Los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones.

2.5.2. Aplicación en el sistema

El `EstudianteController` depende de `IEstudianteDAO`, no de una clase concreta.

- Las implementaciones son inyectadas desde el exterior.

2.5.3. Ventajas obtenidas

- Bajo acoplamiento.
- Mayor facilidad para pruebas y cambios.

6. Conclusión

El SRP es una guía esencial para diseñar sistemas robustos y bien estructurados. Separar responsabilidades permite desarrollar software más claro, escalable ,permite la reutilización del código y de fácil mantenimiento. En el ejemplo, aplicar SRP al CRUD de estudiantes en Java permitió dividir las funciones en tres clases con propósitos definidos y sin acoplamiento.

7. Referencias

- Martin, R.C. (2003). *Agile Software Development, Principles, Patterns, and Practices.*
- Medium - SOLID Principles Explained in Plain English
- Oracle Java Official Documentation

8. Evaluación según rúbrica

Criterio	Puntaje Máximo	Obtenido
Explicación clara de SRP	4	_
Identificación de responsabilidades mezcladas	4	
Ejemplo que cumple SRP correctamente	4	
Análisis de ventajas de aplicar SRP	4	
Presentación clara y ordenada	4	
TOTAL	20	