Universidad de las Fuerzas Armadas ESPE

Departamento: Ciencias de la Computación

Carrera : Ingeniería de Software

Taller académico Nª: 5

---

1. Información General

- **Asignatura:** Análisis y Diseño
- **Apellidos y nombres de los estudiantes:** Joan Cobeña, Juan Pasquel, Ruben Benavides, Edison Verdesoto
- **NRC:** 22426
- **Fecha de realización:** 09/07/2025

- **Asunto**: Análisis y Recomendaciones sobre la Aplicación del Principio de Responsabilidad Única (SRP) en el Desarrollo de CRUD de Estudiantes

---

Aplicación del Principio SOLID

1. Resumen

Los principios de diseño de software SOLID son fundamentales para construir aplicaciones robustas, mantenibles y escalables. Este informe se centra en el **Principio de Responsabilidad Única (SRP)**, el cual estipula que una clase debe tener una y solo una razón para cambiar. A través del análisis de un caso práctico de gestión de estudiantes (CRUD), se demuestra cómo la violación de este principio conduce a un código frágil y de alto acoplamiento.

La correcta aplicación del SRP, separando las responsabilidades de manejo de datos, persistencia y presentación, resulta en un sistema más limpio, flexible y fácil de mantener. Se concluye que la adopción del SRP no es una opción, sino una necesidad estratégica para garantizar la calidad y longevidad del software. Se recomienda la adopción obligatoria del SRP, la capacitación continua del equipo y la integración de su revisión en los procesos de control de calidad del código.

2. Análisis del Problema: Violación del Principio de Responsabilidad Única

En el documento de referencia, se presenta un ejemplo claro de una clase Estudiante que incumple el SRP. Esta clase concentra múltiples responsabilidades, una práctica común pero perjudicial en el desarrollo de software.

2.1. Identificación de Responsabilidades Mezcladas

La clase Estudiante inicial asume tres roles distintos:

1. **Modelo de Datos**: Contiene los atributos del estudiante (id, nombre) y la lógica para su inicialización. Su responsabilidad es representar la entidad "Estudiante".
2. **Persistencia de Datos**: A través del método guardarEnBD(), la clase se encarga de la lógica para interactuar con una base de datos.
3. **Presentación de Datos**: Mediante el método imprimir(), la clase define cómo se debe mostrar la información del estudiante al usuario (en este caso, por consola).

Código que NO cumple con SRP:

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

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

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

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

2.2. Consecuencias Negativas

Esta acumulación de responsabilidades genera problemas significativos:

- **Alto Acoplamiento**: La lógica de negocio está fuertemente ligada a la tecnología de la base de datos y al formato de presentación. Un cambio en la base de datos (ej. de MySQL a MongoDB) o en la interfaz de usuario (ej. de consola a una API REST) obliga a modificar la clase Estudiante, aunque el modelo de datos del estudiante no haya cambiado.
- **Baja Cohesión**: Los métodos dentro de la clase tienen propósitos muy diferentes, lo que hace que el código sea menos comprensible y más difícil de razonar.
- **Dificultad de Mantenimiento y Pruebas**: Probar la clase se vuelve complejo, ya que una prueba unitaria del modelo de datos podría invocar accidentalmente lógica de persistencia o presentación. El mantenimiento se convierte en una tarea arriesgada, donde un cambio puede tener efectos secundarios inesperados.

3. Solución Propuesta: Refactorización y Aplicación de SRP

La solución correcta, como se indica en el documento, es refactorizar el código para que cada clase tenga una única responsabilidad.

3.1. Separación de Responsabilidades

1. **Clase Estudiante (Modelo):** Su única responsabilidad es contener los datos del estudiante. Es un objeto simple (POJO) sin conocimiento de cómo se guarda o se muestra.
```
public class Estudiante {
    private int id;
    private String nombre;
    // Constructor, Getters y Setters...
}
```

2. **Clase EstudianteDAO (Data Access Object):** Su única responsabilidad es la persistencia. Maneja toda la comunicación con la base de datos para la entidad Estudiante.
```
public class EstudianteDAO {
    public void guardar(Estudiante estudiante) {
        // Lógica para guardar en la base de datos...
        System.out.println("Guardando estudiante en la BD...");
    }
}
```

3. **Clase EstudiantePrinter (Vista/Presentación):** Su única responsabilidad es la presentación de los datos del estudiante.
```
public class EstudiantePrinter {
    public void imprimir(Estudiante estudiante) {
```

```
        // Lógica para mostrar los datos...
        System.out.println("Estudiante: " + estudiante.getNombre());
    }
}
```

3.2. Ventajas de la Solución

Esta separación proporciona beneficios directos y medibles:

Si cambias cómo se imprimen los datos, no tocas la clase: Estudiante (Modelo) ni EstudianteDAO

Si cambias cómo se guardan, tampoco tocas el: Estudiante (Modelo) ni EstudiantePrinter

Cada clase tiene UNA SOLA razón para cambiar: Cambios en su funcionalidad guardar para el DAO, imprimir para el Printer de la Vista y creación/definición de atributos el Modelo (Single Responsibility)

- **Mantenibilidad**: Si se necesita cambiar el motor de base de datos, solo se modifica EstudianteDAO. Si se quiere un nuevo formato de reporte, solo se afecta a EstudiantePrinter (o se crea una nueva clase de presentación).
- **Reusabilidad**: La clase EstudianteDAO puede ser utilizada por cualquier componente del sistema que necesite guardar estudiantes, no solo por el flujo inicial.
- **Claridad y Testeabilidad**: Cada clase es pequeña, enfocada y fácil de probar de manera aislada, lo que aumenta la fiabilidad del sistema.

4. Conclusiones

1. El Principio de Responsabilidad Única (SRP) es esencial para evitar la creación de "clases dios" que centralizan demasiada lógica, volviéndose frágiles y difíciles de mantener.
2. La violación del SRP, como en el ejemplo inicial, genera un alto acoplamiento que incrementa los costos y riesgos asociados a cualquier cambio en el software.
3. La refactorización hacia un diseño que respeta el SRP produce un código desacoplado, cohesivo y modular. Este diseño es inherentemente más flexible a los cambios y más sencillo de escalar.
4. La inversión de tiempo en aplicar SRP durante el diseño y desarrollo se traduce directamente en una reducción de la deuda técnica y un ciclo de vida del producto más saludable.

5. Recomendaciones

Con base en las conclusiones anteriores, se recomienda al equipo de desarrollo y a la gerencia

técnica tomar las siguientes acciones:

1. **Establecer SRP como Estándar**: Adoptar formalmente el Principio de Responsabilidad Única como un estándar de codificación obligatorio para todos los nuevos desarrollos y refactorizaciones.
2. **Fomentar la Capacitación**: Realizar talleres y sesiones de formación sobre los principios SOLID para asegurar que todos los miembros del equipo comprendan su importancia y aplicación práctica.
3. **Integrar en Revisiones de Código (Code Reviews)**: Incluir un punto de control específico en el proceso de revisión de código para identificar violaciones del SRP. La pregunta clave debe ser: "¿Cuántas razones para cambiar tiene esta clase?".
4. **Priorizar la Refactorización**: Asignar tiempo dentro de los sprints para refactorizar componentes críticos existentes que no cumplen con el SRP, con el fin de mejorar la mantenibilidad del código base actual.

La implementación de estas medidas fortalecerá nuestras prácticas de ingeniería de software y contribuirá directamente a la construcción de productos de mayor calidad.