**Universidad de las Fuerzas Armadas ESPE**

**Departamento:** Ciencias de la Computación

**Carrera:** Ingeniería de Software

**Informe académico Nª:** 1

**1. Información General**

* **Asignatura:** Análisis y Diseño de Software
* **Apellidos y nombres de los estudiantes:** Pamela Chipe, Carlos Jaya, Elkin Pabón
* **NRC:** 23305
* **Fecha de realización:** 09/07/2025

**2. Objetivo del Informe y Desarrollo**

**Objetivo:**  
Analizar el Principio de Responsabilidad Única (SRP) del conjunto SOLID dentro de un sistema CRUD en Java, con el fin de comprender su impacto en la calidad del diseño orientado a objetos, promoviendo una arquitectura modular, mantenible y escalable que facilite la separación de responsabilidades y reduzca el acoplamiento entre componentes del sistema.

**Desarrollo:**

1. **Introducción Teórica:**

El principio de responsabilidad única (SRP, por sus siglas en inglés: Single Responsibility Principle) establece que una clase debe tener una única razón para cambiar. Esto significa que cada clase debe estar enfocada en una única funcionalidad específica dentro del sistema.

Aplicar SRP permite que el código sea más comprensible, fácil de mantener y menos propenso a errores, ya que evita que una sola clase mezcle diferentes tipos de lógica, como presentación, persistencia o manipulación de datos. Este principio es el primero del conjunto SOLID y sienta las bases para un diseño de software más robusto y escalable.

1. **Actividades Prácticas:**

Se trabajó con un sistema básico de registro de estudiantes (CRUD) programado en Java. Inicialmente, se presentó una clase denominada Estudiante, la cual concentraba múltiples responsabilidades: definía los atributos del estudiante, contenía métodos para guardar en la base de datos y también incluía funciones para imprimir la información por pantalla. Esta estructura violaba el principio de responsabilidad única (SRP), ya que la clase tenía más de una razón para cambiar.

**Situación Inicial:** La clase **Estudiante** original mezclaba varias responsabilidades: definía los atributos del estudiante, incluía métodos para guardar en la base de datos y también funciones para imprimir la información en consola. Esta combinación violaba el Principio de Responsabilidad Única (SRP), generando un fuerte acoplamiento y dificultando el mantenimiento, la extensión y las pruebas del sistema.



**Problemas Detectados:**

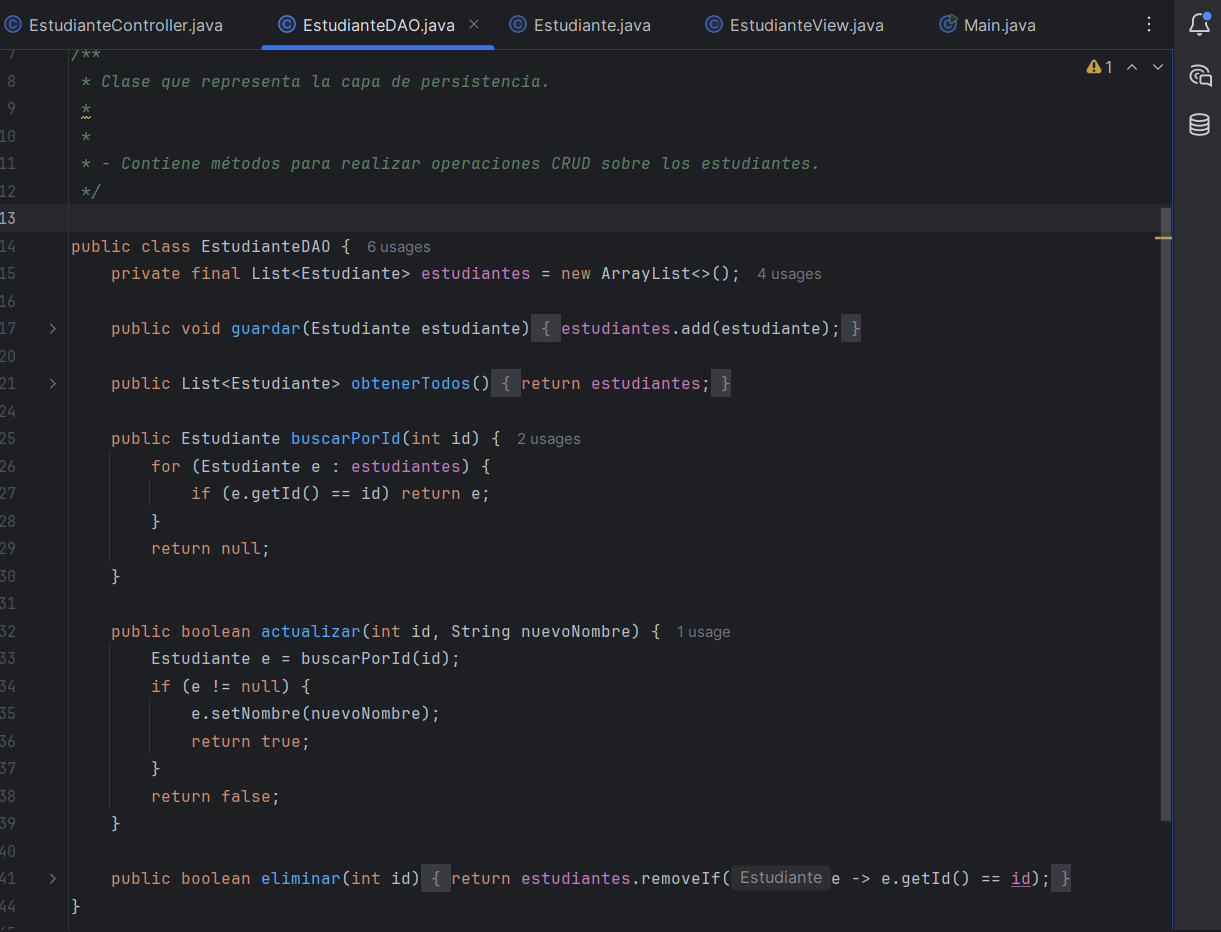
* Cambios en la estructura de datos del estudiante requerían modificar la misma clase.
* Cambios en la persistencia (por ejemplo, pasar de base de datos a archivos) obligaban a alterar la clase modelo.
* Cambios en la presentación (por ejemplo, de consola a HTML) también requerían modificaciones en la clase Estudiante.  
   Esto implicaba múltiples razones para cambiar una sola clase, aumentando el riesgo de errores y dificultando la escalabilidad.

**Proceso de Refactorización:**

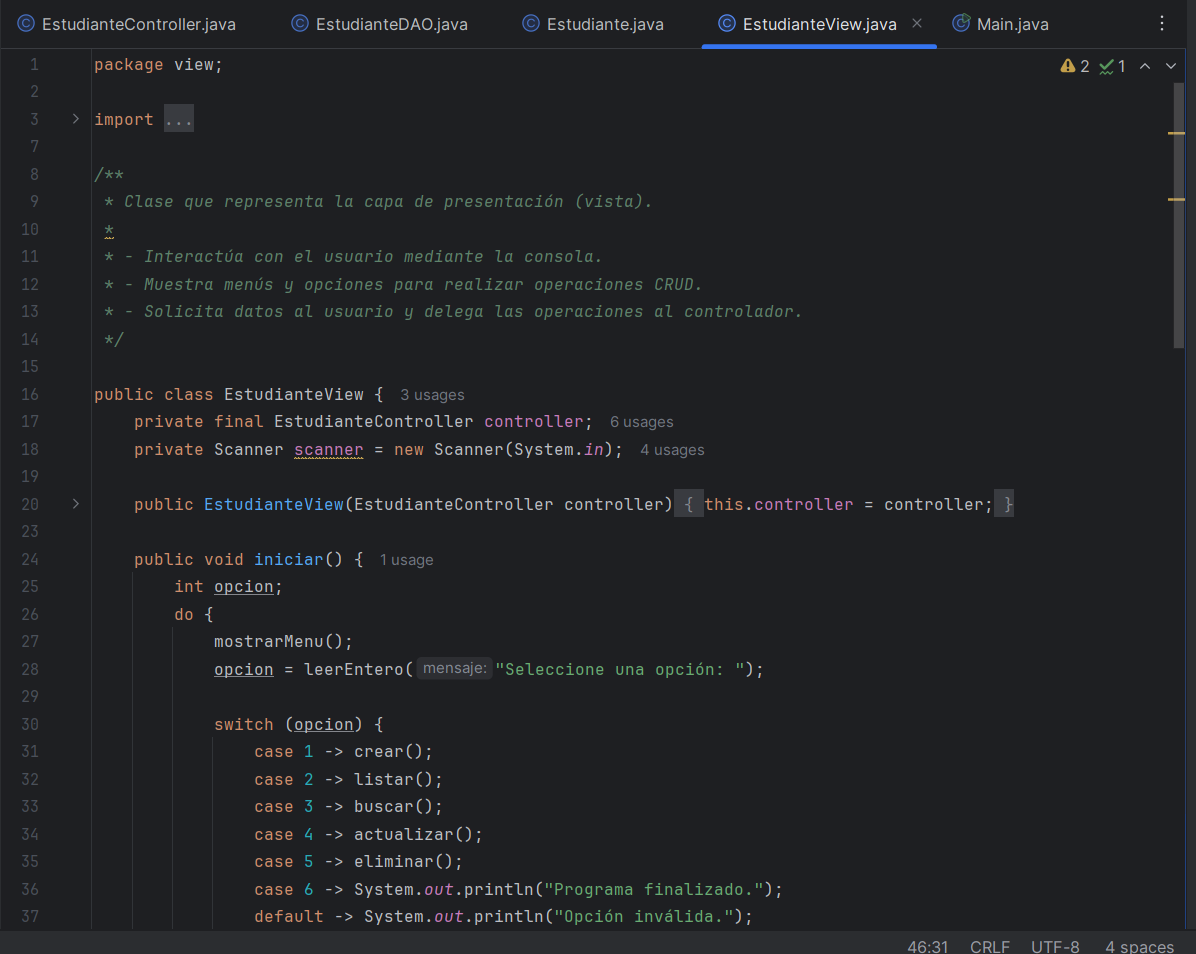
1. **Clase Estudiante (Modelo de Datos):** Se aisló la definición de los atributos (id, nombre) y sus métodos de acceso, eliminando cualquier lógica relacionada con almacenamiento o presentación.



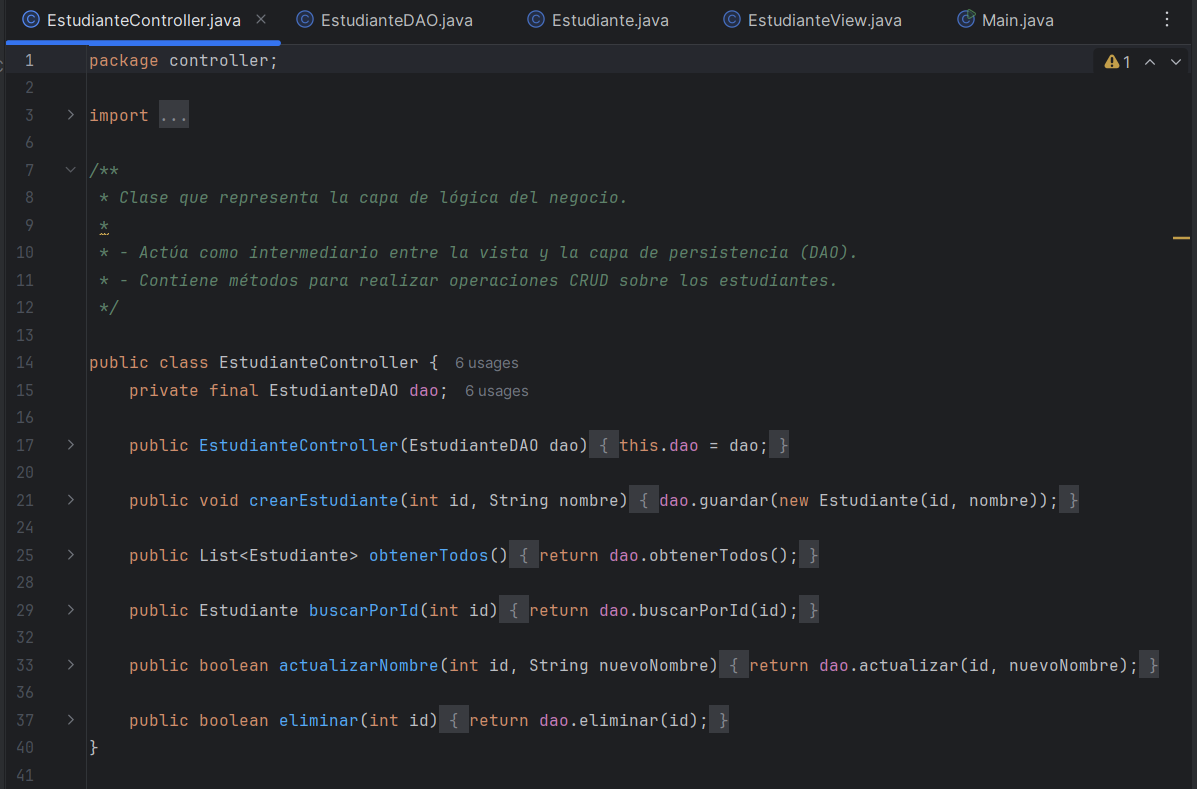
1. **Clase EstudianteDAO (Capa de Persistencia):** Se creó una clase dedicada exclusivamente a la gestión del almacenamiento y recuperación de objetos Estudiante, implementando métodos para guardar, buscar, actualizar y eliminar. Esta separación desacopla la lógica de datos del modelo.



1. **Clase EstudianteView (Capa de Presentación):** Se desarrolló una clase encargada únicamente de mostrar la información del estudiante, permitiendo cambiar la forma de presentación sin afectar el modelo ni la persistencia.



1. **Clase EstudianteController (Capa de Control):** Se añadió una capa intermedia que gestiona la lógica de negocio y coordina la comunicación entre la vista y el modelo, sin incluir lógica de presentación ni de acceso a datos.



**Beneficios Obtenidos:**

* Cada clase tiene una única responsabilidad clara y específica.
* Se redujo el acoplamiento entre las capas del sistema, facilitando su mantenimiento.
* El sistema se volvió más flexible y escalable, permitiendo cambios en una capa sin impactar las demás.
* Se mejora la testabilidad y la posibilidad de reutilización del código.

Esta refactorización permite cumplir con el Principio de Responsabilidad Única (SRP) dentro del patrón arquitectónico MVC, optimizando la estructura y calidad del software.

1. **Discusión de Resultados:**

La aplicación del principio SRP permitió dividir el sistema en capas bien definidas. Cada clase pasó a tener una única función, lo que reduce el acoplamiento y mejora la cohesión del sistema.

Esta mejora estructural no solo ayuda al desarrollador a entender mejor el sistema, sino que también facilita el mantenimiento, las pruebas unitarias y la posibilidad de ampliar funcionalidades sin afectar otras partes del código.

**Tabla comparativa:**

| **Aspecto** | **Antes de SRP** | **Después de SRP** |
| --- | --- | --- |
| **Estructura** | Clase única con múltiples funciones | Clases separadas por responsabilidad |
| **Acoplamiento** | Alto | Bajo |
| **Mantenibilidad** | Difícil de modificar sin afectar otras partes | Cambios localizados por capa |
| **Pruebas** | Complicadas de aplicar | Fáciles y enfocadas por clase |
| **Escalabilidad** | Limitada | Mejor preparada para crecer |

1. **Conclusiones:**

* La aplicación del principio SRP mejora notablemente la calidad del diseño del software, ya que permite que cada clase se enfoque en una única responsabilidad. Esto facilita su comprensión, mantenimiento y evolución, reduciendo el riesgo de errores y conflictos en futuras modificaciones.
* Dividir la lógica del sistema en capas específicas como modelo, persistencia y presentación favorece la construcción de sistemas desacoplados. De esta forma, los cambios en una parte del sistema no afectan otras, lo que permite una evolución más controlada y una arquitectura más profesional.
* El SRP facilita la adopción de arquitecturas limpias y patrones como MVC, al promover una distribución clara y coherente de responsabilidades. Esto permite construir sistemas reutilizables, escalables y más fáciles de extender o adaptar ante nuevas necesidades.

1. **Recomendaciones:**

* Aplicar el principio SRP desde las fases iniciales del desarrollo, asegurando que cada clase tenga una responsabilidad claramente definida. Esto ayuda a evitar estructuras rígidas y facilita la evolución del sistema a medida que crecen los requisitos.
* Utilizar arquitecturas en capas y patrones como MVC o Clean Architecture, ya que estos promueven la separación lógica del sistema en componentes bien definidos. Combinados con SRP, permiten un desarrollo más organizado, mantenible y escalable.
* Incorporar refactorizaciones y revisiones de diseño como parte del proceso de desarrollo, priorizando la aplicación de principios SOLID. Esto garantiza un código más limpio, modular y preparado para adaptarse a nuevas funcionalidades o tecnologías.

**3. Referencias (Norma APA 7.0)**

* Sommerville, I. (2016). Software engineering (10th ed.). Pearson Education.
* Pressman, R. S., & Maxim, B. R. (2014). Software engineering: A practitioner’s approach (8th ed.). McGraw-Hill Education.
* Martin, R. C. (2018). Clean architecture: A craftsman's guide to software structure and design. Pearson Education.