Aplicación de gestión de usuarios utilizando clases avanzadas

Estructura de clases implementada

Clase Persona

- **Atributos: nombre** y **apellido**, que representan los datos básicos de una persona.
- **Métodos: getters** y **setters** para ambos atributos.

Esta clase es la base para otras clases como Usuario, lo que permite reutilizar los atributos comunes y respeta el principio de herencia en POO. La decisión de usar esta clase base hace que el código sea extensible y fácil de mantener.

Clase Permiso

- **Atributos: nombre**, que describe el permiso (por ejemplo, "crear_usuario").
- **Métodos:** Un **constructor** para inicializar el permiso con un nombre.

La clase Permiso permite crear permisos específicos que pueden ser asociados a roles, favoreciendo la reutilización y la flexibilidad del sistema. Es simple y tiene un único propósito, cumpliendo con el principio de responsabilidad única.

Clase Rol

- **Atributos: nombre** (el nombre del rol) y **permisos** (una lista de objetos Permiso asociados a este rol).
- **Métodos: agregar_permiso**, que asegura que un permiso sólo se añade una vez a la lista.

Un rol agrupa varios permisos, lo que permite gestionar los permisos de manera más eficiente. La lista de permisos hace que los roles sean fácilmente escalables y modificables sin necesidad de cambiar el código en otras partes del sistema. Esto sigue el principio de abierto/cerrado, ya que puedes agregar nuevos permisos sin modificar la clase Rol.

Clase Usuario (hija de Persona)

- **Atributos: nombre, apellido** (heredados de Persona), **correo_electronico**, **password** (contraseña encriptada), y **roles** (lista de objetos Rol).
- **Métodos:**
    - **set_password** y **verify_password** para gestionar la autenticación segura con contraseñas encriptadas.
    - **agregar_rol** para asignar roles al usuario.
    - **tiene_permiso** para verificar si el usuario tiene un permiso a través de sus roles.

La clase Usuario es el centro del sistema, ya que representa a los usuarios que se autentican y tienen permisos asignados. La herencia de Persona permite reutilizar atributos comunes como el nombre y apellido. Además, la encriptación de contraseñas con bcrypt es una práctica recomendada para mantener la seguridad del sistema. El método **tiene_permiso** facilita la verificación de permisos de manera centralizada.

Clase SistemaAutenticacion

- **Atributos: usuarios**, lista de objetos Usuario registrados en el sistema.
- **Métodos:**
    - **registrar_usuario**, que agrega usuarios al sistema.
    - **autenticar**, que verifica si un usuario existe y tiene una contraseña correcta.

Esta clase centraliza la lógica de autenticación, asegurando que solo los usuarios registrados con contraseñas correctas puedan acceder al sistema. La gestión de usuarios en una lista hace que el sistema sea sencillo y escalable.

Justificación de las decisiones de diseño

En primer lugar, se usa la herencia para separar las responsabilidades y compartir atributos comunes entre clases (por ejemplo, Usuario hereda de Persona). Esto hace que el código sea modular y reutilizable.

Para garantizar la seguridad, se implementa la encriptación de contraseñas con bcrypt para proteger la información sensible y evitar que las contraseñas sean almacenadas en texto claro.

Además, la estructura de roles y permisos permite un sistema flexible, donde los permisos se pueden asignar a los roles y luego los roles a los usuarios. Esto

simplifica la gestión de accesos y permite que los permisos se manejen de manera eficiente. Cada clase tiene una única responsabilidad, lo que permite agregar más funcionalidades (como nuevos permisos o roles) sin afectar las clases existentes. Además, el sistema es fácilmente ampliable para incluir nuevas características sin necesidad de reescribir mucho código.

Por último, me he asegurado de que el diseño siguiera varios principios de diseño de software, como:

- ★ **Responsabilidad única:** cada clase tiene un único propósito.,
- ★ **Abierto/cerrado**: las clases son fáciles de extender sin modificar el código existente
- ★ **Dependencia inversa** (los módulos de alto nivel no dependen de los de bajo nivel, como se ve en el uso de roles y permisos).