Sistema de Gestión de Rangos Militares

Objetivo: Desarrollar un sistema de gestión de rangos militares que permita simular la administración de personal militar utilizando los principios de la programación orientada a objetos en Java.

Descripción del Proyecto:

Los estudiantes deberán crear un programa por interfaz gráfica que gestione un conjunto de soldados. **Este proyecto cambiará un poco la lógica del anterior proyecto, lean bien.**

Requisitos:

1. **Clases y Objetos:**
    - Definir una clase base `Soldado` que incluya atributos como `nombre`, `ID` y `rango`. Esta clase deberá tener métodos para mostrar información del soldado.
2. **Herencia:**
    - ☐ **Clase Base: `Soldado`**
    - **Atributos:**
        - **`String nombre`: Nombre del soldado.**
        - **`String id`: Identificador único del soldado.**
        - **`String rango`: Rango del soldado (puede ser un valor por defecto que se sobreescriba en las clases derivadas).**
    - **Métodos:**
        - **`void mostrarInformacion()`: Muestra la información del soldado.**
        - **`void patrullar()`: De acuerdo al rango, imprimirá una forma de patrullar u otra(sea creativo)**
        - **`void saludar()`: Si su nombre comienza y termina por la misma letra, saludara de cierta manera, sino de otra.**
        - **`void regañado()`: Cuando no cumpla su deber, bajara su nivel del rango en una unidad. Si su nivel de rango es 1, en lugar de bajar a 0 será expulsado (sacado de la lista donde este)**
        -
    - ☐ **Clase Abstracta: `Rango`**
    - **Atributos:**
        - **`int nivel`: Nivel jerárquico del rango (del 1 a 4, donde 1 es el menor rango, 4 es el máximo).**
    - **Métodos Abstractos:**
        - **`abstract void realizarAccion()`: Define acciones específicas para cada rango (casi todos pusieron prints, sean creativos!)**
    - ☐ **Clases Derivadas:**
        a. **`SoldadoRaso`**

- **Hereda de: `Soldado`**
- **Atributos:**
  - **No tiene atributos adicionales, pero su rango se inicia como "Soldado Raso".**
- **Métodos:**
  - **`@Override void realizarAccion()`: Implementa una acción específica, como seguir órdenes.**

b. `Teniente`

- **Hereda de: `Rango`**
- **Atributos:**
  - **`String unidad`: Unidad a la que pertenece el teniente.**
- **Métodos:**
  - **`@Override void realizarRegaño()`: Este rango permite regañar a otro soldado**

c. `Capitan`

- **Hereda de: `Rango`**
- **Atributos:**
  - **`int cantidadSoldadosBajoSuMando`: Número de soldados bajo su mando.**
- **Métodos:**
  - **`@Override void realizarAccion()`: Puede coordinar misiones o liderar estrategias.**

d. `Coronel`

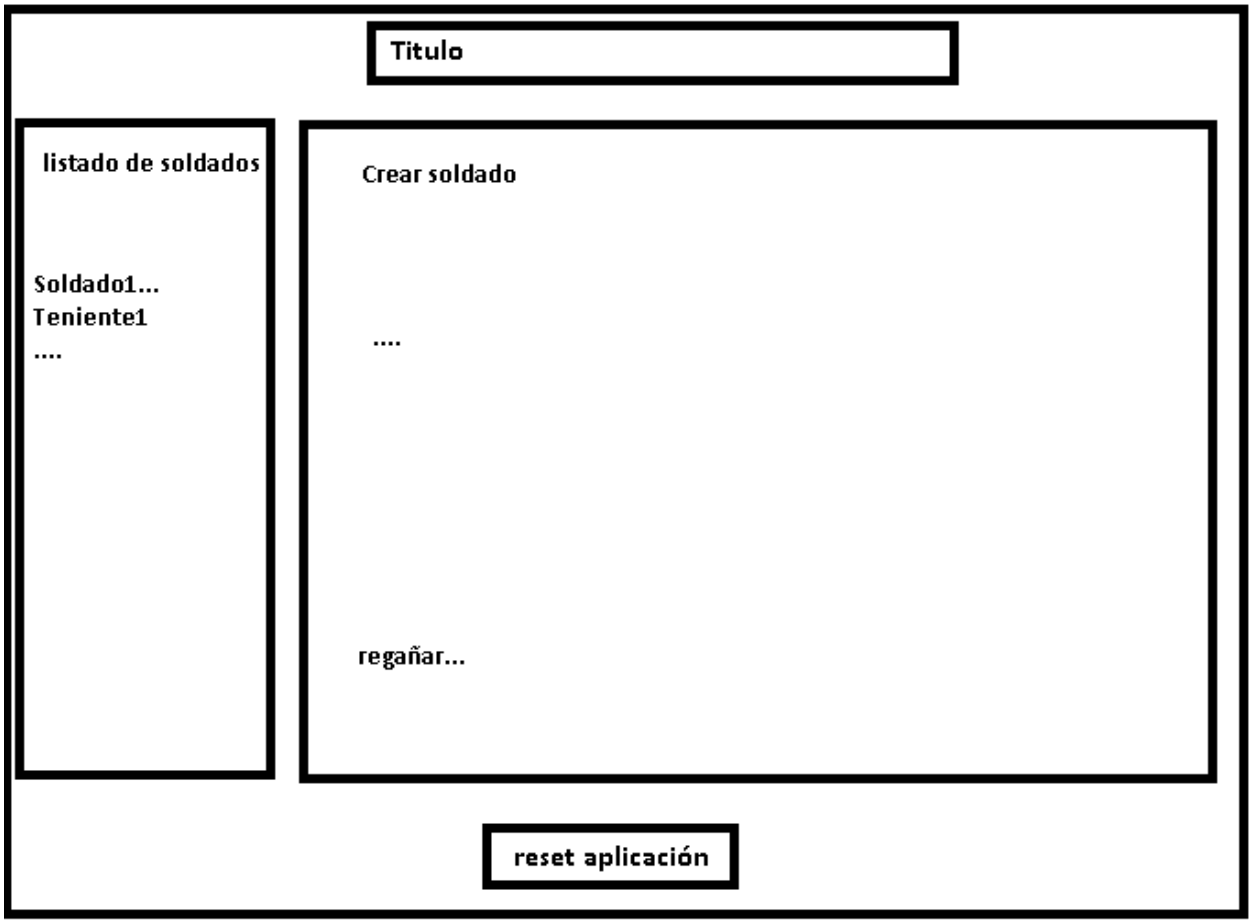
- **Hereda de: `Rango`**
- **Atributos:**
  - **`String estrategia`: Estrategia militar que está implementando.**
- **Métodos:**
  - **`@Override void saludar()`: Sobreescribe el método por defecto y siempre saluda de forma épica**

Resumen de la Herencia

- **`Soldado` es la clase base que contiene los atributos comunes a todos los soldados, como `nombre` e `id`.**
- **`Rango` es una clase abstracta que establece un nivel jerárquico común para todos los rangos, además de definir un método abstracto que cada rango específico debe implementar.**

- **Las clases derivadas como `SoldadoRaso`, `Teniente`, `Capitan` y `Coronel` heredan de `Soldado` y `Rango`, cada una implementando sus propios atributos y comportamientos específicos.**
    -

3. **Interfaces:**
    - Crear una interfaz `OperacionesMilitares` que declare los siguientes métodos:
        - `void asignarMision(String mision)`: Asigna una misión al soldado.
        - `void reportarEstado()`: Reporta el estado actual del soldado.
    - Cada clase de rango debe implementar esta interfaz, ofreciendo su propia lógica para los métodos. Por ejemplo, un `Capitan` podría reportar más información que un `SoldadoRaso`.
4. **Clases Abstractas:**
    - Definir una clase abstracta `Rango` que contenga atributos y métodos comunes para todos los rangos. Esta clase puede incluir:
        - Un atributo `nivel` que indique la jerarquía del rango.
        - Un método abstracto `void realizarAccion()` que será implementado por cada clase derivada para definir acciones específicas según el rango (p. ej., un `Coronel` puede ordenar estrategias, mientras que un `SoldadoRaso` puede seguir instrucciones).
    - Las clases derivadas de `Rango` deberán implementar este método abstracto, proporcionando su propia lógica.
5. **Polimorfismo:**
    - Demostrar polimorfismo mediante la sobreescritura de métodos en las clases de rango y el uso de referencias de la clase base para acceder a los objetos de rango. Por ejemplo, tener un método que reciba un `Soldado` y llame a `reportarEstado()`, mostrando el comportamiento específico de cada rango.
6. **Modificadores:**
    - Utilizar `static` para atributos o métodos que deben ser compartidos entre todas las instancias (por ejemplo, un contador de soldados).
    - Emplear `final` en atributos que no deben cambiar una vez asignados (como el ID del soldado).
7. **Paquetes:**
    - Organizar el código en paquetes adecuados (por ejemplo, `militar.soldados`, `militar.misiones`, `militar.rangos`).
8. **Sobrescritura y Downcasting:**
    - Incluir ejemplos de sobrescritura de métodos en las clases derivadas y mostrar cómo realizar downcasting de una referencia de la clase base a una clase derivada. Por ejemplo, un método que recibe un `Soldado` y, al verificar su tipo, puede hacer un downcast a `Capitan` para acceder a métodos específicos.



La aplicación debe tener esta estructura. Debe contar con un listado por defecto de soldados de todos los tipos. Un botón de reset para deshacer todos los cambios y volver a los valores por defecto. En la mitad se deben estar los botones, checkbox, radiobuttons y demás ítems de acción para hacer todas las operaciones de todos los soldados previamente dicho.

Entrega: Los estudiantes deberán presentar su código bien documentado y hacer una demostración por interfaz gráfica (no se acepta por consola) del funcionamiento del sistema, mostrando cómo se pueden gestionar diferentes rangos militares y realizar operaciones asignadas.

Evaluación: Los detalles porcentuales por rúbrica son: C.E. 4 (70%), C.E. 14 (5%) C.G. 4 (15%), C.G.2 (10%)

C.E. 14: La evaluación se basará en la correcta aplicación de los conceptos de programación orientada a objetos, la claridad del código, la documentación y la efectividad del sistema presentado.

C.G. 4: Con la evaluación del trabajo en github. Se evalúa el nivel individual de commits (si no hace commits se entiende que no participó en la codificación), que usen las ramas, los proyectos, los issues y todo lo visto

C.G.2: ***Debe investigar por cuenta propia e implementar en el proyecto:***

a. Implementar el look and feel en la interfaz
b. Investigar como hacer un menu en la interfaz e implementarlo en esta aplicación

---

NOTA IMPORTANTE: LOS INTEGRANTES DEBEN IR EN EL README.TXT, EL QUE NO APAREZCA EN EL ESE ARCHIVO NO SE LE PONDRÁ NOTA

¡Esperamos que disfruten del reto y aprendan mucho sobre programación orientada a objetos con este ejercicio práctico!