TAREA PRÁCTICA SOBRE LA UNIDAD 6: JERARQUÍA DE PERSONAJES

Las actividades de esta tarea se van a centrar en la implementación de una clase llamada `Personaje`, donde vamos a utilizar una gran parte de los mecanismos y recursos relacionados con el desarrollo de avanzado de clases (herencia y composición de clases, redefinición y ampliación de métodos, implementación de interfaces, polimorfismo, ligadura dinámica, etc.).

Se trata de llevar a cabo la implementación de un **pequeño programa para probar el comportamiento básico de una jerarquía de clases basada en los personajes de un juego de rol**. La aplicación permitirá crear objetos de distintas subclases de la clase `Personaje` con unas características determinadas. **La clase** `Personaje` **será abstracta**, es decir, no podrá ser instanciada, pero incluirá una serie de características (atributos) y comportamientos (métodos) comunes para todas las clases que hereden de ella.

1. Implementa la clase **`Personaje`** donde se definan los atributos mínimos para representar las siguientes características de un personaje:

- **Nombre** del personaje (texto).
- **Descripción** del personaje (texto con una descripción algo más detallada).
- **Energía** del personaje (número real entre 0 y 100), que indicará lo "saludable" que se encuentra el personaje en cada momento.

Esas características serán comunes a todo personaje, ya sea un soldado, mago, caballero, etc. Sin embargo no podrán crearse objetos de tipo `Personaje` por ser una clase demasiado "genérica" a la que le faltan aún algunas propiedades y métodos más específicos. Por eso será una clase **abstracta**. Ahora bien, aunque se trate de una clase abstracta, dispondrá de algunos métodos que habrá que implementar.

Tanto la clase `Personaje` como todas sus subclases y las interfaces que éstas implementen deberán estar agrupadas en un **paquete** llamado **`juego.personajes`**.

2. Implementa los siguientes métodos para la clase **`Personaje`**:

- **Constructores**. La clase constará de dos constructores: un **constructor básico con un único parámetro** (el **nombre** del personaje) y un **constructor con dos parámetros** (**nombre** y **descripción**). En el primer caso la descripción se rellenará automáticamente con el texto "*Sin descripción*". Para evitar la repetición de código, este primer constructor debe llamar al segundo constructor (uso de la llamada a `this()`), que será quien realmente lleve a cabo las labores de asignación e inicialización de valores. En ambos casos, el personaje que se construya tendrá una **energía inicial de 100,0**.
- Métodos `get` para los atributos **nombre**, **descripción** y **energía** (`getNombre()`, `getDescripcion()`, `getEnergia()`). No habrá métodos `set`.
- Método `toString()`, que generará una cadena de caracteres con las propiedades del personaje con el siguiente formato: ***{nombre: XXX; descripcion: YYY; energía: ZZZ}***, donde XXX será el nombre del personaje; YYY la descripción; y ZZZ la energía (con dos decimales). Este método nunca será llamado desde un programa principal, pues no se podrán instanciar objetos de esta clase, pero sí deberá ser llamado desde los métodos `toString()` de clases que hereden de de ella. De esta forma evitaremos la redundancia de código (la generación de la cadena de caracteres con los atributos nombre, descripción y energía se

hará una sola vez en todo el código del proyecto). Para ello habrá que hacer llamadas a `super.toString()`.

Esta clase será la clase padre de otras cuatro clases hijas que heredarán de ella sus características y comportamientos principales. Esas clases hijas serán los cuatro tipos de personajes que va a contemplar nuestro prototipo: **soldados**, **caballeros**, **magos** y **escuderos**.

3. Define la interfaz `Hablador`.

La clase **`Personaje`**, además de todo lo anterior, también implementa la interfaz `Hablador`, la cual incorpora dos métodos: `saludar()` y `despedirse()`. La clase **`Personaje`** no implementará ninguno de estos métodos. No tiene por qué hacerlo dado que es abstracta, pero cualquier método instanciable que herede de ella tendrá que implementarlos y cada una lo implementará de una manera diferente (**polimorfismo**). La forma de saludo o despedida de cada tipo de personaje habrá que implementarla por tanto en cada clase específica.

4. Define la interfaz `Entrenable`.

En nuestro prototipo de juego también existirá la interfaz llamada `Entrenable`. Toda clase que implemente esta interfaz debe incluir un método llamado `entrenar(double porcentaje)` que permitirá a cada tipo de personaje (caballero, mago, etc.) mejorar en sus características específicas (fuerza, magia, etc.). Ahora bien, no todos los personajes serán entrenables, de manera que esta interfaz no será implementada por la clase **`Personaje`**, sino sólo por algunas de sus subclases. El parámetro que se le pasa al método `entrenar` es el porcentaje de la energía actual del personaje que se desea invertir en su entrenamiento. Esto hará que tras el entrenamiento el personaje habrá mejorado en algunas de sus características específicas pero habrá perdido ese porcentaje de energía. Esa cantidad, obviamente tendrá que estar obligatoriamente en el rango 0-100 (es un porcentaje). Si no es así, se debería lanzar una excepción de tipo `IllegalArgumentException` con el mensaje de error "***Error: Porcentaje de entrenamiento no válido***". Cada personaje implementará de una manera específica su algoritmo de entrenamiento (**polimorfismo**).

5. Implementa la clase `Soldado`.

La clase `Soldado` estará caracterizada por las siguientes propiedades (atributos y comportamientos):

- Todas aquellas que herede por el hecho de ser una subclase de **`Personaje`**.
- **Puntos de fuerza** (número real positivo), que le servirán para luchar mejor o peor durante los combates.
- **Puntos de escudo** (número real positivo), que le servirán para defenderse con mayor o menor soltura durante los combates.
- Todas aquellas que dependan de implementar la interfaz `Entrenable`.

El saludo de un soldado será del tipo "*Saludos del soldado XXX. A sus órdenes*", donde XXX es el nombre del soldado. La despedida será de la forma "*El soldado XXX se despide*".

Tanto la clase `Soldado` como el resto de **las clases que hereden de `Personaje` tendrán siempre dos constructores**: uno más básico, en este caso sólo con el nombre del soldado, y otro con algún parámetro más, en este caso añadiendo también la descripción del soldado. **El constructor más básico siempre deberá llamar al constructor más complejo** para evitar la redundancia de código. Para ello tendrás que utilizar una llamada a `this()`. Además de eso, **el constructor más complejo**

debe siempre "aprovechar" todo lo que haga el constructor de su clase padre (en este caso **`Personaje`**) para evitar nuevamente la redundancia de código. Para ello tendrás que utilizar una llamada a `super()`. Esto tendrás que hacerlo en los constructores de todas las subclases de la clase **`Personaje`**.

Un soldado recién creado tendrá 100,00 de energía, 0,00 de puntos de fuerza y 0,00 de puntos de escudo. Recuerda que cualquier asignación debe hacerse solamente en uno de los dos constructores y el otro debe "aprovecharse" para reutilizar ese código (llamada a `this()`).

En el caso del método `toString()` habrá que hacer también algo similar para todas las clases que hereden de **`Personaje`**: habrá que "aprovechar" lo que genera el método `toString()` de la clase padre (generación de la cadena con **nombre**, **descripción** y **energía**) y a partir de ahí construir la nueva cadena con los atributos específicos de la subclase. Para ello tendrás que hacer una llamada a `super.toString()`. En el caso de la clase `Soldado` habría que añadir los atributos de **puntos de fuerza** y **puntos de escudo**.

Para la clase `Soldado`, el método `toString()` generará una cadena de caracteres con las propiedades del personaje con el siguiente formato: ***{nombre: XXX; descripcion: YYY; energía: ZZZ; fuerza: FFF; escudo: EEE}***, donde XXX será el nombre del personaje; YYY la descripción; ZZZ la energía (con dos decimales); FFF los puntos de fuerza (con dos decimales) y EEE los puntos de escudo (con dos decimales).

Por el hecho de implementar la interfaz `Entrenable`, la clase `Soldado` también tendrá que incorporar el método `entrenar`. En este caso el **algoritmo de entrenamiento** será el siguiente:

- Se suma a los **puntos de fuerza** actuales el porcentaje de energía que se ha decidido invertir en el entrenamiento (parámetro del método).
- Se suma a los **puntos de escudo** actuales el porcentaje de energía que se ha decidido invertir en el entrenamiento. Es decir, lo mismo que se hace para los puntos de fuerza.
- Se resta a la **energía** actual el porcentaje de energía que se ha decidido invertir en el entrenamiento.

Ejemplos:

- Si intentamos entrenar a un soldado recién creado (100 de energía y 0 de fuerza y escudo) con un 20% de energía (`entrenar (20.0)`), el soldado debería terminar con una fuerza de 20, un escudo de 20, y una energía de 80 (el personaje ha perdido un 20% de su energía al entrenarse).
- Si intentamos entrenar a ese mismo soldado, que ahora tiene 80 de energía, 20 de fuerza y 20 de escudo, con un 50% de energía (`entrenar (50.0)`), el soldado debería terminar con una fuerza de 60 (20 + el 50% de 80), un escudo de 20 (20 + el 50% de 80), y una energía de 40 (el personaje ha perdido un 50% de su energía al entrenarse).
- Si volviéramos a entrenar nuevamente a ese mismo personaje con un 20% de su energía (`entrenar (20.0)`), entonces se nos quedaría con una fuerza de 68 (60 + el 20% de 40), un escudo de 68 (60 + el 20% de 40), y una energía de 32 (40 menos el 20% de 40).

Ten en cuenta que el parámetro del método `entrenar` es un porcentaje de la energía que tenga el personaje en cada momento. **¡No es un número absoluto!** Y por tanto la llamada a `entrenar (20.0)` no siempre va a tener el mismo efecto. Dependerá de la energía actual del personaje. Y por supuesto, al tratarse de un porcentaje, siempre deberá estar entre 0 y 100.

6. Implementa la clase `Caballero`.

La clase `Caballero` estará caracterizada por las siguientes propiedades (atributos y comportamientos):

- Todas aquellas que herede por el hecho de ser una subclase de **`Personaje`**.
- **Puntos de fuerza** (número real positivo), que le servirán para luchar con mayor o menor destreza durante las batallas.
- **Puntos de escudo** (número real positivo), que le servirán para defenderse mejor o peor durante las batallas.
- Todas aquellas que dependan de implementar la interfaz `Entrenable`.

Como puedes observar, tendrá **los mismos atributos que la clase** `Soldado`, aunque es posible que sus métodos implementen comportamientos diferentes.

El saludo de un caballero será del tipo "*El caballero XXX os saluda*", donde XXX es el nombre del caballero. La despedida será de la forma "*El caballero XXX se despide*".

Un caballero recién creado tendrá, al igual que un soldado, 100,0 de energía, 0,0 de puntos de fuerza y 0,0 de puntos de escudo.

Para la clase `Caballero`, el método `toString()` generará una cadena de caracteres con las propiedades del personaje con el siguiente formato: ***{nombre: XXX; descripcion: YYY; energía: ZZZ; fuerza: FFF; escudo: EEE}***, donde XXX será el nombre del personaje; YYY la descripción; ZZZ la energía (con dos decimales); FFF los puntos de fuerza (con dos decimales) y EEE los puntos de escudo (con dos decimales).

La clase `Caballero` también implementará la interfaz `Entrenable`, por lo que también tendrá que incorporar el método `entrenar`. En este caso el **algoritmo de entrenamiento** para un caballero será el siguiente:

- Se suma a los **puntos de fuerza** actuales el porcentaje de energía que se ha decidido invertir en el entrenamiento (parámetro del método) multiplicado por 1,5. Es decir, que el entrenamiento de un caballero es un 50% más eficiente que el de un soldado de infantería.
- Se suma a los **puntos de escudo** actuales el porcentaje de energía que se ha decidido invertir en el entrenamiento también multiplicado por 1,5. Lo mismo que sucede con los puntos de fuerza.
- Se resta a la **energía** actual el porcentaje de energía que se ha decidido invertir en el entrenamiento.

Ejemplos:

- Si intentamos entrenar a un caballero recién creado (100 de energía y 0 de fuerza y escudo) con un 20% de energía (`entrenar (20.0)`), el caballero debería terminar con una fuerza de 30 (0 + el 20% de 100 * 1,5), un escudo de 30, y una energía de 80 (el personaje ha perdido un 20% de su energía al entrenarse).
- Si intentamos entrenar a ese mismo caballero, que ahora tiene 80 de energía, 30 de fuerza y 30 de escudo, con un 50% de energía (`entrenar (50.0)`), el caballero debería terminar con una fuerza de 90 (30 + el 50% de 80 * 1,5), un escudo de 90 (30 + el 50% de 80 por 1,5), y una energía de 40 (el personaje ha perdido un 50% de su energía al entrenarse).
- Si volviéramos a entrenar nuevamente a ese mismo personaje con un 20% de su energía (`entrenar (20.0)`), entonces se nos quedaría con una fuerza de 102 (90 + el 20% de 40 * 1,5), un escudo de 102 (90 + el 20% de 40 * 1,5), y una energía de 32 (40 menos el 20% de 40).

7. Implementa la clase `Mago`.

La clase `Mago` estará caracterizada por las siguientes propiedades (atributos y comportamientos):

- Todas aquellas que herede por el hecho de ser una subclase de **`Personaje`**.
- **Puntos de magia** (número real positivo), que le servirán para poder llevar a cabo hechizos de mayor o menor calidad.
- Todas aquellas que dependan de implementar la interfaz `Entrenable`.

En este caso tiene atributos específicos diferentes a los de las clases `Soldado` y `Caballero`, y es muy probable que algunos de sus métodos también implementen comportamientos diferentes.

Un mago recién creado tendrá 100,0 de energía y 0,0 de puntos de magia.

El saludo de un mago será del tipo "*Te encuentras ante el mago XXX. Póstrate ante mi poder...*", donde XXX es el nombre del mago. La despedida será de la forma "*El mago XXX se despide. Buena suerte y que la magia te acompañe*".

Para la clase `Mago`, el método `toString()` generará una cadena de caracteres con las propiedades del personaje con el siguiente formato: ***{nombre: XXX; descripcion: YYY; energía: ZZZ; magia: MMM}***, donde XXX será el nombre del personaje; YYY la descripción; ZZZ la energía (con dos decimales) y MMM los puntos de magia (con dos decimales).

Los magos también implementarán la interfaz `Entrenable`. Su **algoritmo de entrenamiento** es el siguiente:

- Se suma a los **puntos de magia** actuales el porcentaje de energía que se ha decidido invertir en el entrenamiento (parámetro del método).
- Se resta a la **energía** actual el porcentaje de energía que se ha decidido invertir en el entrenamiento.

8. Implementa la clase `Escudero`.

La clase `Escudero` estará caracterizada por las siguientes propiedades (atributos y comportamientos):

- Todas aquellas que herede por el hecho de ser una subclase de **`Personaje`**.
- **Puntos de escudo** (número real positivo), que le servirán para defenderse.
- **Referencia al caballero al cual sirve**. Obligatorio. Todo caballero debe servir a un escudero. Si no, no puede ser creado.

Si te fijas, todo escudero debe servir a un caballero y por tanto debe tener una referencia a un caballero que ya exista. Eso significa que cuando se cree un escudero (llamada al constructor) habrá que pasarle una referencia a un objeto de tipo `Caballero`. Si no se le pasa un caballero que ya exista, no se podrá instanciar un objeto de tipo `Escudero`. Los dos constructores de esta clase tendrán que incorporar ese parámetro en su lista de parámetros:

- `public Escudero (String nombre, Caballero caballero)`
- `public Escudero (String nombre, String descripcion, Caballero caballero)`

Si el caballero que se le pasa al constructor no es correcto, el objeto no podrá ser creado y se lanzará una excepción de tipo `IllegalArgumentException` con el mensaje "***Error: Parámetros de creación del personaje no válidos. Un escudero debe tener caballero asignado***".

Un escudero recién creado tendrá 100,0 de energía y 0,0 de puntos de escudo.

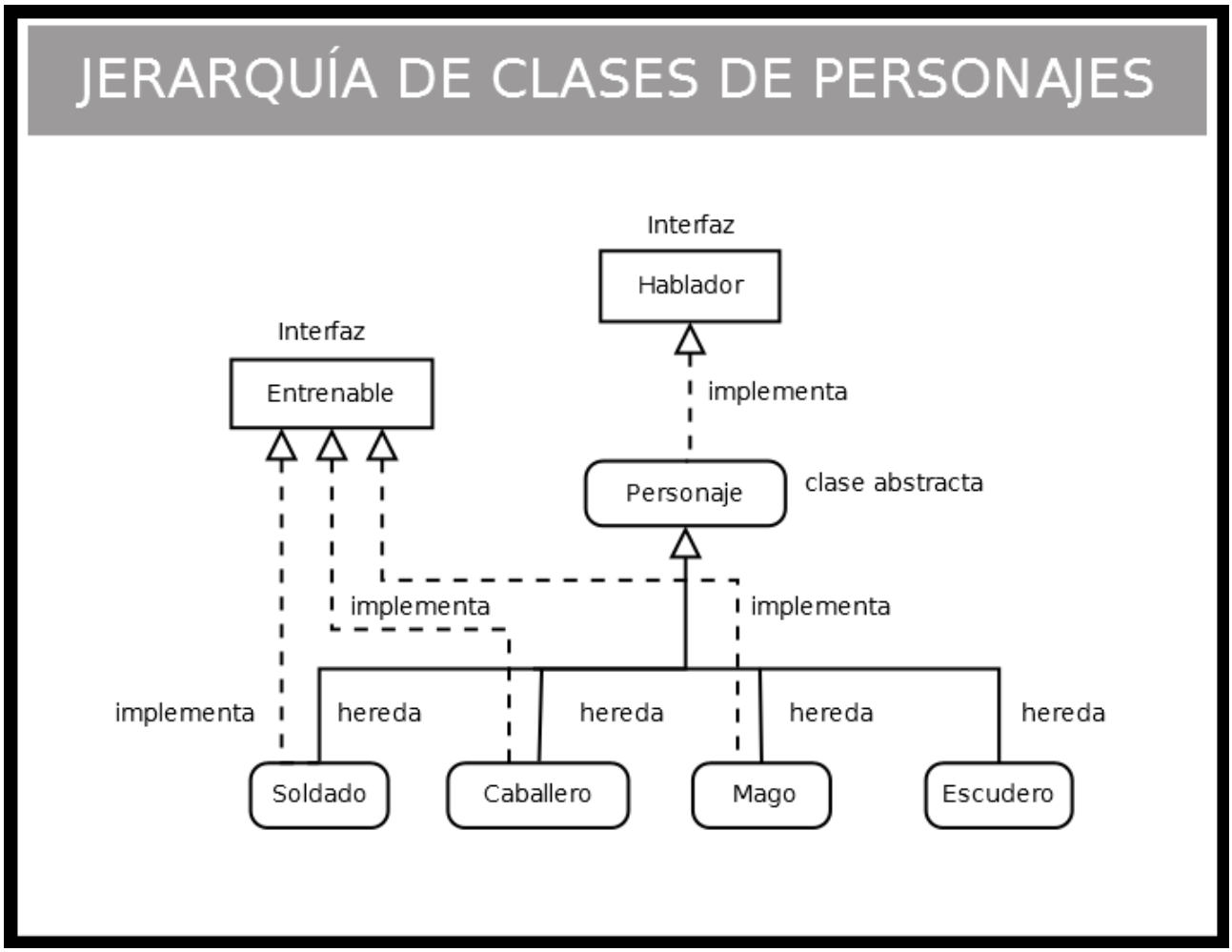
El saludo de un escudero será del tipo "*Hola. Os saluda el escudero XXX al servicio del caballero*

YYY.", donde XXX es el nombre del escudero e YYY el nombre del caballero al que sirve. La despedida será simplemente "*Hasta luego.*".

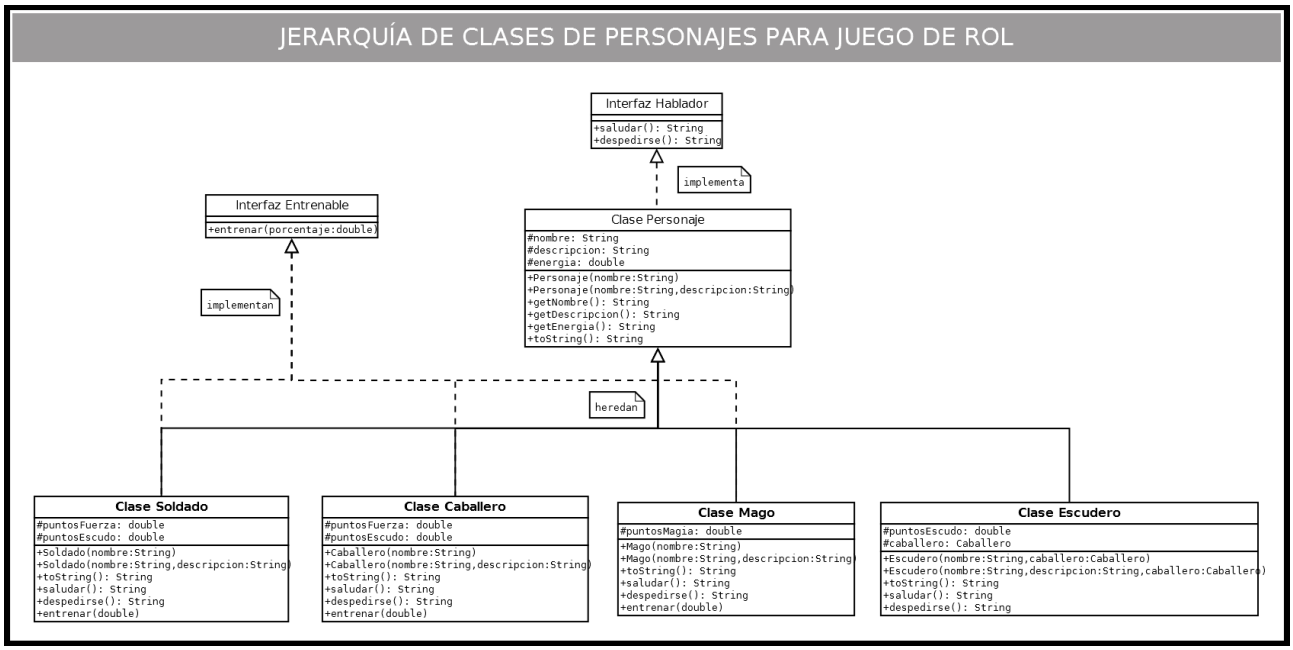
Para la clase `Escudero`, el método `toString()` generará una cadena de caracteres con las propiedades del personaje con el siguiente formato: ***{nombre: XXX; descripcion: YYY; energía: ZZZ; escudo: EEE; caballero: CCC}***, donde XXX será el nombre del personaje; YYY la descripción; ZZZ la energía (con dos decimales); EEE los puntos de escudo (con dos decimales) y CCC el nombre del caballero al cual sirve.

Los escuderos no implementan la interfaz `Entrenable`, así que **no disponemos de algoritmo de entrenamiento**.

Una vez descrita toda la jerarquía de clases e interfaces, podríamos plasmarla mediante el siguiente diagrama:



Un diagrama más exhaustivo y detallado podría ser:



En el primer diagrama aparecen únicamente clases e interfaces y sus relaciones entre ellas. En el segundo, más exhaustivo, aparecen también atributos y métodos En este segundo diagrama se representan todos aquellos métodos que es necesario redefinir, ampliar o definir desde cero. Aquéllos para que los que no sea necesario sobrescribir ni redefinir (como por ejemplo `getNombre()`), no aparecerán en la descripción de la clase específica, pues asumirán el método de su superclase como propio sin hacer ningún tipo de especialización.

9. Documenta apropiadamente el código de tus clases e interfaces con **comentarios javadoc** para poder generar una documentación correcta:

- **Documentación a nivel de clase**. Debes incluir un buen comentario para documentar la clase o la interfaz en general.
- **Documentación a nivel de métodos**, incluyendo, por cada método:
  - **Descripción del método**. Descripción "corta" y descripción detallada.
  - **Descripción de los parámetros** (`@param`).
  - **Descripción del valor devuelto** (`@return`), si es que devuelve algo.
  - **Descripción de las posibles excepciones que puede lanzar** (`@throws`).