Práctica: RPG

Vamos a modelar, de manera muy básica, la estructura de un videojuego RPG. Los objetivos de la práctica son los siguientes:

- Definir personajes: nombre, raza y profesion.
- La raza y la profesión han de influir en las características de los personajes.
- Permitir que los personajes puedan recibir daño y curarse.
- Que los jugadores puedan recoger objetos.
- Uso de items consumibles: comida, pociones...
- Permitir equiparse con objetos.
- Habilitar la posibilidad de subir de nivel.

Vamos a hacer un uso extensivo de herencia e interfaces para que el código sea extensible, es decir, que se puedan añadir elementos nuevos fácilmente.

Como el proyecto va a tener un número considerable de clases, interfaces y enums, será necesario organizar el proyecto de forma correcta.

(1) Vamos a crear en primer lugar un package llamado **Character** que albergará todo lo relacionado con el personaje.

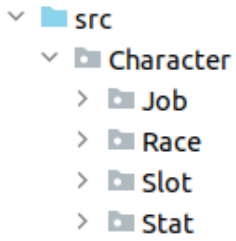

Figure 1: Character package structure

1. Características

Una característica representará cada uno de los atributos (físicos o mentales) del personaje. Estas características tendrán un valor que define lo desarrollado que un personaje tiene cada atributo. En nuestro caso usaremos 4:

- Strength (Fuerza): define el poder físico del personaje. Influirá principalmente en el valor de ataque físico, pero también define el peso que un personaje puede llevar.
- Dexterity (Destreza): representa la coordinación ojo-mano, la agilidad, reflejos o equilibrio. También influirá en la capacidad de esquivar un ataque.
- Constitution (Constitución): define la vida y aguante de un jugador.
- Intelligence (Inteligencia): representa la fácilidad del personaje para aprender y razonar. Determina el poder mágico.

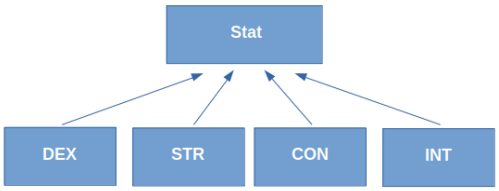

Figure 2: Stat structure

(2) En primer lugar, crea un nuevo package llamado **Stat** dentro del package **Character**.

(3) Crea una clase abstracta llamada **Stat** que representará una característica genérica. **Stat** tendrá un único atributo llamado **value**, cuyo valor recibirá por parámetro en el constructor. La clase stat tendrá los siguientes métodos:

```
//Devuelve el valor actual de la característica
public int getValue(){
```

```
    ...
}
//Aumenta el valor de la característica en 1
public void increase(){
    ...
}
//Disminuye el valor de la característica en 1
public void decrease(){
    ...
}


@Override
//Devuelve el nombre simple de la clase
public String toString(){
   ...
}
```

(4) Una vez implementada esta clase crea las 4 subclases descritas anteriormente (Strength, Dexterity, Constitution e Intelligence).

(5) Crea una batería de test (como has aprendido en Entornos) y comprueba que lo que has implementado funciona correctamente.

2. Razas

Las razas representan la naturaleza de los personajes, es decir, sus características genéticas.

- Human (Humano): Raza equilibrada.
- Orc (Orco): Raza imponente físicamente.
- Elf (Elfo): Destacan por su inteligencia y destreza.

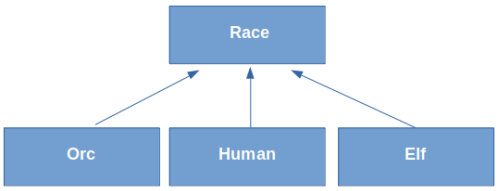

Figure 3: Race structure

Para implementar las razas seguiremos una estructura similar a la de las características.

(6) Como ya hemos hecho anteriormente, creamos un package llamado **Race** dentro del package **Character**.

(7) Crea una clase genérica (abstract) llamada **Race** que tendra los siguientes métodos:

```
//Devuelve el modificador de la profesión segun el stat
public abstract int modifier(Stat stat);

@Override
//Devuelve true si son la misma clase
public boolean equals(Object obj){
    ...
}

@Override
//Devuelve el nombre simple de la clase
public String toString(){
    ...
}
```

(8) Crea las 3 razas descritas anteriormente (Humano, Orc, Elf) como subclases de la clase abstracta **Race**.

Los modificadores de cada una de las razas son los siguientes (la suma de los bonus ha de dar 5):

- Orc: Strength +5, Constitution +3, Intelligence -3
- Human: Strength +2, Constitution +2, Dexterity +1
- Elf: Dexterity +3, Intelligence +3, Constitution -1

```
//Para saber que el tipo de stat puedes usar instanceof
 if (stat instanceof Dexterity)
```

(9) Crea 2 razas de tu elección que modifiquen los stats que consideres oportunos.

(10) Comprueba que cada raza aplica los modificadores correctamente mediante test.

3. Profesiones

Las profesiones definen el modo de vida principal del personaje y, por consiguiente, influye en sus características. Las profesiones que vamos a implementar son las siguientes:

- Warrior (Guerrero): centrado en el combato cuerpo a cuerpo, es diestro en el uso de todo tipo de armas. Su característica principal es la fuerza.
- Mage (Mago): su principal característica es la inteligencia. Ataques mágicos letales a distancia.
- Assassin (Asesino): se basa en el sigilo para acercarse a su objetivo. La destreza es su principal característica.

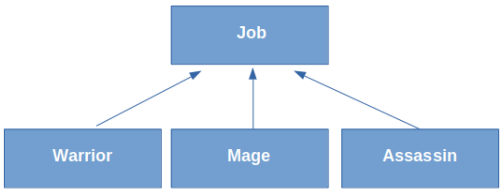

Figure 4: Job structure

Para implementar las profesiones seguiremos una estructura igual que en las razas.

(11) Creamos un package llamado **Job** dentro del package **Character**.

(12) Crea una clase genérica (abstract) llamada **Job** que tendra los siguientes métodos:

```
//Devuelve el modificador de la profesión segun el stat
public abstract int modifier(Stat stat);

@Override
//Devuelve true si son la misma clase
public boolean equals(Object obj){
    ...
}

@Override
//Devuelve el nombre simple de la clase
public String toString(){
    ...
}
```

(13) Crea las 3 profesiones descritas anteriormente (Warrior, Mage, Assassin) como subclases de la clase abstracta **Job**.

Los modificadores de cada una de las profesiones son los siguientes (repartimos 5 puntos):

- Warrior: Strength +3, Constitution +2
- Mage: Intelligence +4, Dexterity +1
- Assassin: Dexterity +3, Strength +1, Constitution +1

(14) Crea 2 profesiones de tu elección que modifiquen los stats que consideres oportunos.

(15) Comprueba que cada profesión aplica los modificadores correctamente usando test.

4. Personaje

(16) Crea la clase **Character** que representará a un jugador dentro del juego. Crea la clase dentro del package **Character**.

Esta clase recibirá tres parámetros por constructor: name (**String**), race (**Race**) y job (**Job**). Además recibirá, también por constructor, un atributo de cada una de las características (**Stat**) creadas en uno de los puntos anteriores (**Strength**, **Dexterity**, **Constitution**, **Intelligence**). Puedes iniciar estas características con el valor 5.

La clase ha de implementar los siguientes métodos:

```
public String getName(){
    ...
}

public Race getRace(){
    ...
}

public Job getJob(){
    ...
}

//(Valor base Dexterity + bonif. raza + bonif.profesion)*2
public double velocity(){
    ...
}

//(Valor base Strength + bonif. raza + bonif.profesion)*2
public double power(){
    ...
}

//(Valor base Intelligence + bonif. raza + bonif.profesion)*2
public double magic(){
    ...
}

@Override
//Muestra toda la información de un personaje
public String toString(){
    ...
}
```

La salida del método toString() ha de ser parecida a la siguiente:

> My name is Caliel. I'm an elf assassin My stats are: Strength: 6 Dexterity: 11 Constitution: 5 Intelligence: 8 Velocity: 22.0 Power: 12.0 Magic: 16.0 Health: 125.0

(17) Añade test a tu bateria de test que aseguren del correcto funcionamiento de la clase.

Una parte importante, y divertida, de los videojuegos es gestionar la vida de nuestros personajes. Para ello hemos de hacer que el personaje tenga vida, pueda recibir daño y también curarse. Como es algo que un personaje, y muchos otros elementos pueden hacer, vamos a modelar esta cualidad a través de una interfaz.

(18) Crea una interfaz llamada **IDamageable** dentro del package **Character**.

La interfaz tendrá los siguientes métodos:

```
public interface IDamageable {
    //(Valor base Constitution + bonif. raza + bonif.profesion)*25
    double maxHealth();
    //Devuelve el valor de vida actual
    double health();
    //Devuelve true si el daño es mayor o igual a la vida
    boolean isDead();
    //Aumenta el daño recibido
    void receivesDamage(double amount);
    //Disminuye el daño recibido. El daño mínimo es 0
    void heals(double amount);
}
```

(19) Haz que la clase **Character** implemente la interfaz **IDamageable**.

(20) Para saber lo que está pasando, añade un sout dentro del método receivesDamage y heals para que muestre una salida parecida a la siguiente:

> Kram received 35.0 damage. Health: 215.0/250.0
>
> Kram healed 5.0 life. Health: 220.0/250.0

(21) Crea los test necesarios para comprobar que los nuevos métodos añadidos funcionan correctamente.

5. Comida

Una de las características habituales de los RPG es poder comer para recuperar vida. Vamos a añadir esta funcionalidad a nuestro prototipo.

Aunque los personajes van a consumir comida no es algo que pertenezca al package **Character**, así que vamos a crear otro package llamado **Item** que usaremos para organizar todo lo relacionado con objetos del juego.

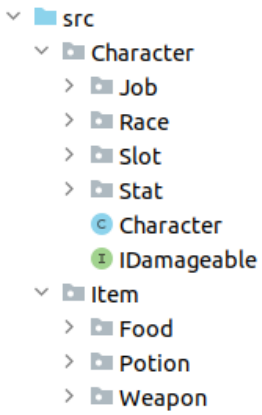

Figure 5: Item Package

(22) Crea el package **Item**

Ya podemos empezar a crear la estructura de clases que van a tener la comida. Puedes echarle un vistazo en la siguiente figura:

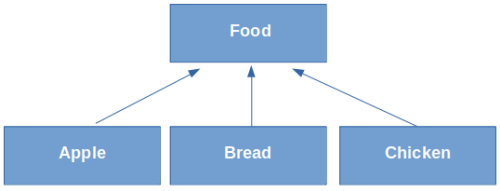

Figure 6: Food structure

Como puedes ver la estructura es la misma que en las ocasiones anteriores. Una clase padre (superclase) abstracta que nos sirve de punto de partida para desarrollar todas las clases de comida necesarias.

(23) Crea la clase abstracta **Food**. Esta clase únicamente tendrá un atributo llamado **power** que representará el "poder"" de la comida.

Aunque queremos que el personaje pueda consumir comida, podemos hacer una

abstracción y extraer una nueva interfaz a la que podemos llamar **IConsumable** que representará a los objetos que se pueden consumir.

La interfaz tendrá el siguiente aspecto:

```
public interface IConsumable {
    //Pasamos el personaje para poder actuar sobre él
    void consumedBy(Character character);
}
```

(24) Haz que **Food** implemente la interfaz **IConsumable** y rellena el cuerpo de la función en la clase abstracta, ya que todas la comida tienen el mismo efecto, curar al personaje.

(25) Crea las 3 comidas del esquema

- Apple: tiene poder 5
- Bread: tiene poder 10
- Chicken tiene poder 25

(26) Crea otras 2 comidas y dales el poder que consideres oportuno.

(27) Añade a la clase **Character** la funcionalidad de poder consumir consumibles.

```
public void consumes(IConsumable consumable) {
    ...
}
```

(28) Para saber lo que está pasando, añade un sout dentro del método consumes que muestre una salida parecida a la siguiente:

Kram consumed: Apple

(29) Crea los test necesarios para comprobar que los nuevos métodos añadidos funcionan correctamente.

(Opcional) Decorator pattern

En ocasiones queremos implementar una versión un poco diferente de un objeto, cambiando un poco su comportamiento. Imaginad que queremos introducir en el juego el concepto de comida envenenada que, cuando se consume, en lugar de curar al personaje le hace daño (el personaje no vería la diferencia visualmente obviamente).

Para conseguir este objetivo podríamos seguir diferentes estrategias. La más obvia seria crear dos versiones de cada una de las comidas (manzana y manzana

envenanada) pero esto nos llevaría a una explosion de clases. Pensad que, añadiendo únicamente el concepto de envenenado, duplicamos el número de clases (manzana y manzana envenenada). Si además queremos añadir otros conceptos, como puede ser comida mejorada (x2 de poder), pasariamos a tener 4 veces más clases (manzana, manzana envenenada, manzana mejorada y manzana envenenada mejorada), algo que a todas luces no es aceptable. Otra opción podría ser introducir attributos bandera (flags) en la manzana, pero esto nos obligaria a estar modificando todas las comidas cada vez que introducimos un nuevo concepto (violando el open-closed principle) además de estar "ensuciando" la clase con atributos extra.

Exite un patrón de diseño que nos ofrece una solución elegante a este problema permitiendonos añadir funcionalidad a un objeto en tiempo de ejecución ("decorar").

Mira el siguiente enlace: Decorator Pattern.

(30) Aplica este patrón y añade las clases **PoisonousFoodDecorator** (en lugar de curar hace daño) y **EnhancedFoodDecorator** (el poder de la comida es el doble).

(31) Crea los test necesarios para asegurarte que todo funciona.

6. Pociones

Usa los conocimientos que has adquirido hasta el momento para implementar por ti mismo las pociones del juego. Se requieren las siguientes pociones:

- Minor Healing Potion: 25 de poder.
- Healing Potion: 50 de poder.
- Greater Healing Potion: 100 de poder.

(32) Implementa las pociones en el juego teniendo en cuenta todo lo que has aprendido hasta ahora. Pregunta al profesor una vez hecha la implementación.

(33) Crea los test necesarios para asegurarte que todo funciona.

Continuará. . .