



Práctica 1 (Programación orientada a objetos)

Una compañía de videojuegos quiere comprobar el equilibrio entre los personajes de su próximo juego de lucha. Nos envía documentación relativa al conjunto P de N personajes del juego (PJs) y a la lógica del combate para que programemos distintas simulaciones y evaluemos si siempre resulta ganador un PJ de la misma clase. La documentación incluye varios ficheros de prueba con personajes ya definidos (prueba1.txt, prueba2.txt, prueba3.txt) y un main.py con código para abrir y leer los ficheros.

Tareas

1. Modelado de la jerarquía de clases de personajes y de objetos que puedan portar (Figura 1):

- La clase abstracta Avatar tiene los siguientes atributos comunes a todos los personajes: name (string), life (integer), strength (integer), defense (integer), weapon (clase Weapon, inicializado a None) y armor (clase Armor, inicializado a None). Deberán implementarse los métodos get() y set() para dichos atributos.
- Existen dos métodos abstractos, correspondientes a las funciones attack() y defend(), disponibles para todos los personajes:
 - def attack(self): int - Devuelve el número de unidades de daño que el personaje causa al atacar.
 - def defend(self): int - Devuelve el número de unidades de protección del personaje ante un ataque.
- Los personajes se dividen en Melee o Caster, subclases abstractas de Avatar. Los Melee atacan cuerpo a cuerpo y pueden portar un arma únicamente del tipo Sword, mientras que los Caster necesitan *maná* para sus acciones y pueden portar únicamente un arma de tipo Wand.
- Los personajes pertenecen a una de las dos clases concretas: Warrior o Mage.
- La especificación de los métodos de cada clase se incluye en el Anexo I.

2. Implementar un programa modular main.py para procesar el fichero prueba1.txt. El programa debe adaptarse para crear los personajes correspondientes. Un ejemplo de ejecución desde la consola sería: `python main.py prueba1.txt` (o desde Spyder, con el fichero main.py seleccionado, *Run -> Configuration per file, Run file with custom configuration, General settings* y en *command line options* añadir prueba1.txt).

3. Ejecutar 30 simulaciones usando los personajes especificados en el fichero prueba1.txt y realizar un análisis de las estadísticas del combate. En cada simulación, codificar la lógica de programa:

1. Seleccionar aleatoriamente dos personajes: un atacante y un defensor.
2. Calcular los puntos de ataque y defensa de cada personaje, hallar la diferencia y reducir la vida del defensor en ese número. Si queda sin vida, se elimina del conjunto P y no es seleccionable.
3. Con una probabilidad del 50%, generar: (i) un objeto de tipo Weapon (de manera equiprobable perteneciente a las subclases Sword y Wand) y (ii) un objeto de tipo Covering (Armor o Shield de manera equiprobable). En ambos casos, el valor del atributo de estos objetos será un valor aleatorio entre 1 y 5. Los objetos generados se le asignarán al atacante, siempre y cuando: (i) sea capaz de portar ese tipo de objeto y (ii) el objeto generado tenga mejores atributos que el objeto que porte actualmente.
4. Repetir el proceso hasta que solo quede un personaje con vida.

Al final de las 30 simulaciones, se pide indicar por pantalla, en orden **descendente**: (i) el número total de veces que ha ganado cada PJ, (ii) el daño medio causado por cada PJ y su desviación típica, (iii) el número de veces que ha ganado cada clase y (iv) el daño medio por cada clase y su desviación típica.

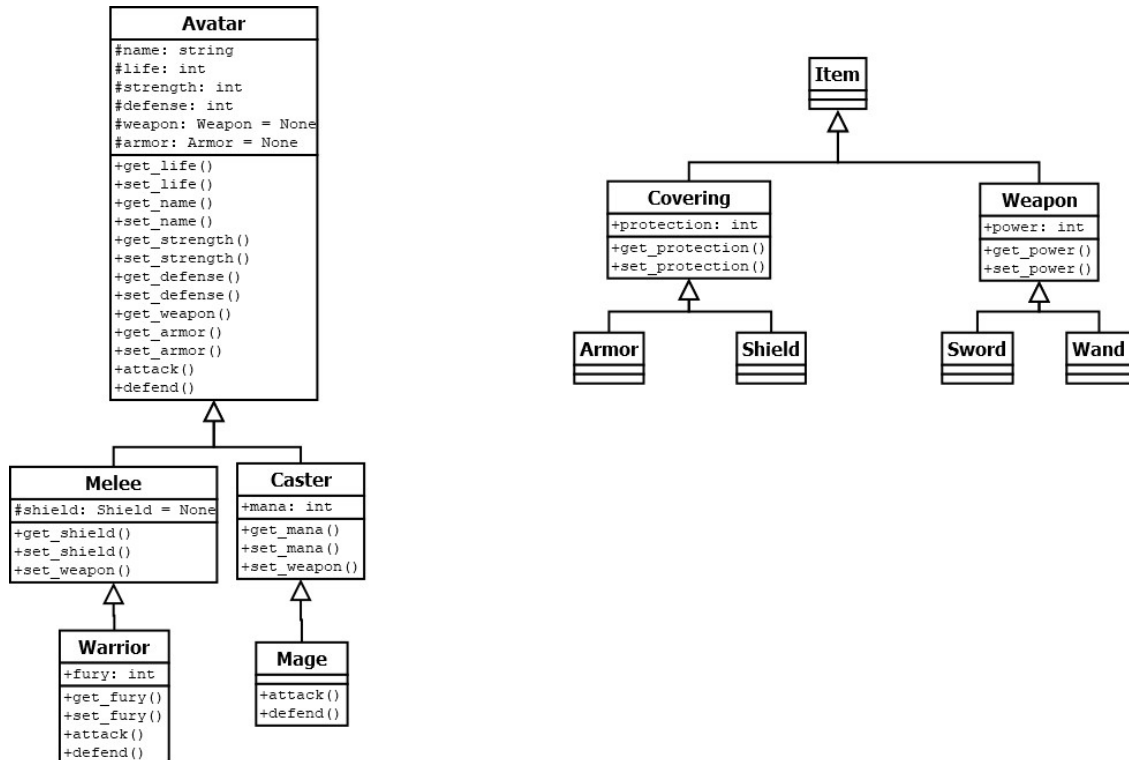


Figura 1. Jerarquía base de clases del videojuego

4. Añadir la clase Priest, que además de las existentes, tenga una nueva acción heal() (Anexo I):

def heal(self): int- Devuelve las unidades con las que un personaje puede incrementar su vida.

Modificar el programa para que, durante la simulación, si el PJ atacante es un Priest, el 75% de las veces atacará, y el otro 25% se curará a sí mismo. Ejecutar la simulación sobre prueba2.txt y obtener estadísticas de la sanación media realizada por cada PJ de clase Priest, así como su desviación típica.

Entrega

Se entregará un archivo zip que contendrá el código fuente y una memoria en formato pdf. :

Todas las clases/funciones del código deben disponer de docstrings de acuerdo al formato del Anexo II (objetivo de la clase/función, parámetros recibidos y valor devuelto). La falta de documentación penalizará un 10% sobre la nota final.

La memoria **no puede exceder las 2 páginas** (con tipo de letra Calibri, Arial o Times New Roman de 11 puntos) e incluirá:

- Manual del usuario *conciso y explicativo* sobre cómo se ejecuta el programa; y
- Descripción *concisa y explicativa* de las fases de desarrollo realizadas.

La falta de alguno de estos apartados penaliza un 10% la nota final por cada apartado.

IMPORTANTE: En cada archivo del código fuente y en la primera página del pdf se indicará el nombre y correo electrónico (@udc.es) de los miembros del grupo de prácticas.

Fecha de entrega: viernes, 10/03/2023 a las 20:00. **Dónde:** en el apartado de Prácticas de Moodle

Quién entrega: sólo uno de los miembros de la pareja.

ANEXO I

Melee

- `set_weapon(w)`: Comprobará que el objeto `w` asignado como valor del atributo `weapon` sea de tipo `Sword`¹.
- `set_shield(w)`: Comprobará que el objeto `w` asignado como valor del atributo `shield` sea de tipo `Shield`¹.

Caster

- `set_weapon(w)`: Comprobará que el objeto `w` asignado como valor del atributo `weapon` sea de tipo `Wand`¹.

Warrior:

- `attack(self)`: Devuelve la suma de: (i) el valor del atributo `strength`, (ii) el valor del atributo `power` (ataque) del objeto `weapon` que porte el guerrero y (iii) un valor aleatorio entre `[0,fury]`.
- `defend(self)`: Devuelve la suma: (i) el valor del atributo `defense`, (ii) el valor del atributo `protection` del objeto `armor` que porte el guerrero, y (iii) el atributo `protection` del objeto `shield`.

Mage:

- `attack()`: Con una probabilidad del 50%, este método incrementa en 2 unidades el valor del atributo `mana`. El daño causado será la suma de: (i) el atributo `strength` y (ii) el valor de ataque del objeto `weapon` que porte el mago, en caso de que el `mana` es mayor que 1, o 1 unidad de daño en otro caso. Finalmente, esta función debe disminuir el valor del atributo `mana` en 1 unidad (`mana` nunca estará por debajo de 0 unidades).
- `defend()`: Devuelve el valor correspondiente la suma de: (i) el valor del atributo `defense` y (ii) el valor del atributo `protection` del objeto `armor` que porte el mago.

Priest:

- `attack()`: La misma implementación que el método `attack()` de la clase `Mage`.
- `defend()`: La misma implementación que el método `defend()` de la clase `Mage`.
- `heal()`: Con una probabilidad del 50%, este método incrementa en 2 unidades el valor del atributo `mana`. Las unidades de sanación será la suma de (`atributo strength + el valor de ataque del objeto weapon`)/2, si `mana` es mayor que 2, o 0 unidades de curación en otro caso. Finalmente, esta función debe disminuir el valor del atributo `mana` en 2 unidades (`mana` nunca estará por debajo de 0 unidades).

¹ Se puede usar de la función `isinstance` para ello.

ANEXO II

Docstrings para clases Python

```
class Clase:
    """Una línea de resumen.

    Descripción en varias líneas

    Attributes
    -----
    attr1 : tipo
        Descripción.
    attr2 : tipo
        Descripción.

    Methods
    -----
    metodo1(param1):
        Una línea de resumen.
    """

    def __init__(self, attr1, attr2):
        """Asigna atributos al objeto.

        Parameters
        -----
        attr1 : tipo
            Descripción.
        attr2 : tipo
            Descripción.

        Returns
        -----
        None.

        """

    def metodo1(self):
        """Una línea de resumen.

        Parametros
        -----
        param1 : tipo
            Descripción.

        Devuelve
        -----
        str
            Resultado de...
        """
```

Docstrings para funciones Python

```
def functionA(param1):
    """Una línea de resumen.

    Parameters
    -----
    param1 : tipo
        Descripción.

    Returns
    -----
    Type.

    """
```