UNIVERSIDADE DA CORUÑA

Grado en Inteligencia Artificial

Programación II – Curso 2023/24

Práctica 1 (Programación orientada a objetos)

Una reconocida compañía tecnológica nos ha solicitado implementar una pequeña prueba de concepto para un videojuego al que van a denominar *Pokemon*[1]. El juego soportará la batalla entre dos entrenadores y sus criaturas *Pokemon*. La batalla tiene lugar hasta que todos los *Pokemon* de un entrenador son derrotados, dando como ganador al contrario. Nos piden una implementación de una pequeña demo, que se dividirá en 3 fases.

FASE 1. Implementación de la jerarquía de clases que nos permita instanciar distintos tipos de *Pokemon*, usando el paradigma de la programación orientada a objetos.[2] Se pide definir una clase abstracta (Pokemon) y una serie de clases hijas concretas para instanciar *Pokemon* de distinto tipo: *agua* (clase WaterPokemon), *fuego* (clase FirePokemon) y *planta* (clase GrassPokemon). La jerarquía, atributos y métodos a implementar se indican como parte de la Figura 1. Dichas clases se definirán dentro de un módulo denominado **pokemon.py**.

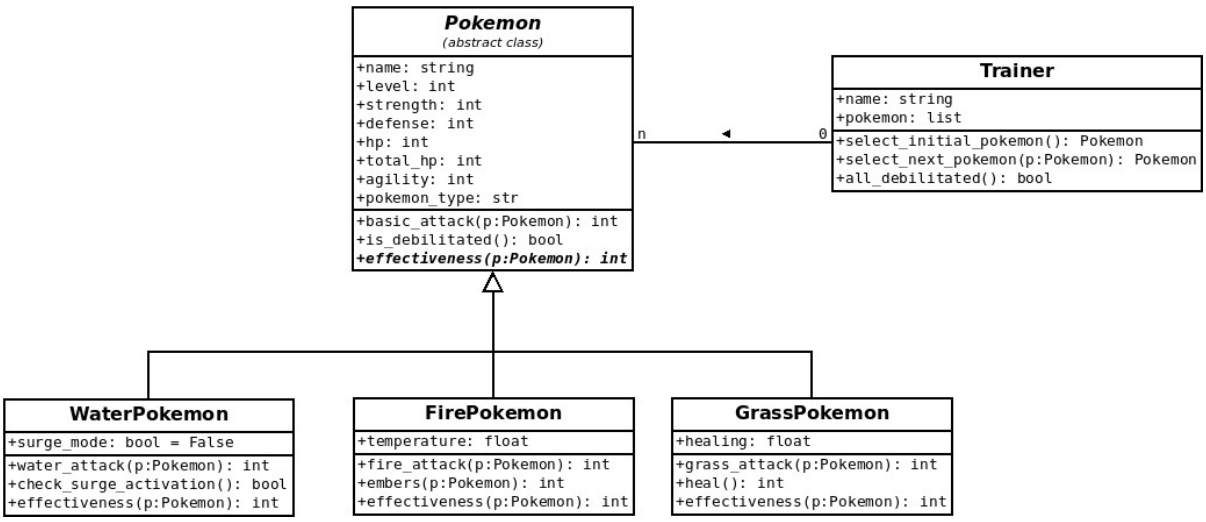

Figura 1: Diagrama de clases con la jerarquía de los distintos tipos de *Pokémon* y sus *Entrenadores*.

La clase abstracta Pokémon implementará los *atributos comunes* (a todos los tipos de Pokémon): nombre (name, de tipo string), tipo de pokemon (pokemon_type, de tipo str), nivel (level, de tipo int), fuerza (strength, de tipo int), defensa (defense, de tipo int), puntos de salud (hp, de tipo int), puntos de salud total (total_hp, de tipo int), y agilidad (agility, de tipo int). El atributo tipo de pokemon (pokemon_type, de tipo str) es también un atributo común, pero su valor específico se definirá directamente dentro del constructor de cada subclase sin pasarlo como parámetro en la función

[1] Pokémon: acrónimo del concepto japonés Poketto Monsuta (Monstruos de Bolsillo), se refiere a la famosa franquicia de videojuegos, donde los jugadores asumen el papel de Entrenadores Pokemon con el objetivo de capturar, entrenar y luchar con criaturas llamadas Pokemon, únicos por habilidades, características y tipos.

[2] La programación orientada a objetoscon Python obliga a que cualquier método definido dentro de una clase reciba como primer parámetro el término **self**, que representa la instancia de la clase que está llamando a dicho método, y es la única forma para acceder a otros atributos y métodos de dicha instancia.

constructora; y se le asignará el valor de 'Water', 'Fire' o 'Grass', según corresponda. Además, se implementarán los métodos de acceso (getters) y modificación (setters[3]) para dichos atributos. Estos métodos se definirán obligatoriamente utilizando los decoradores @property y @setter, siguiendo un *estilo Pythonic* (ver ejemplo en property_example.py en la carpeta con los materiales de esta práctica).

La clase abstracta Pokémon implementará los *métodos comunes* a todos los tipos de *Pokémon*:

- basic_attack(opponent: 'Pokemon') -> int
  Funcionamiento: Disminuye el valor del atributo hp del oponente en *n* unidades de daño, calculadas como el máximo entre 1 y la diferencia entre el atributo strength del atacante y el valor del atributo defense del *Pokemon* oponente.
  Devuelve: El número de unidades de daño que el *Pokemon* causó a p.

- is_debilitated() -> bool
  Devuelve: True si el atributo hp tiene valor cero, y False en otro caso.

Además, incluirá el *método abstracto* effectiveness(opponent: 'Pokemon') -> int, implementado en las clases hijas (ver ANEXO I), y redefinirá el método mágico __str__() para que al mostrar por pantalla una instancia de un objeto *Pokemon* devuelva un string con información en el formato:

```
{name} ({pokemon_type}) Stats: Level: {level}, ATT: {strength}, DEF: {defense}, AGI: {agility}, HP: {hp/total_hp}.
```

donde los valores entre {} indican los valores de los atributos del *Pokémon*[4].

En la jerarquía *Pokémon* existirán las siguientes *clases concretas*: WaterPokemon, FirePokemon y GrassPokemon, que se usarán para crear instancias de distintos tipos de *Pokémon* (ver ANEXO I con la especificación concreta para la implementación de los métodos de cada una de ellas).

Para comprobar de manera automática la correcta implementación de los aspectos básicos de la jerarquía de clases y sus métodos, se proporciona un archivo de prueba, unit_tests_pokemon.py. Se ejecuta desde la línea de comandos con python unit_tests_pokemon.py -v, o directamente desde Spyder para conocer qué casos de test están causando problemas.

FASE 2. Implementar un módulo que defina una clase Trainer incluida en un módulo trainer.py (Figura 1). La clase Trainer dispone de dos *atributos*: el nombre del entrenador (name, de tipo string) y una lista con todos los Pokémon de dicho entrenador (pokemon, de tipo lista). Deberán definirse sus métodos getter y setters con los decoradores @property y @setter correspondientes, así como implementar el resto de sus métodos (Figura 1), cuya especificación recoge el ANEXO II.

FASE 3. Implementar la lógica de la batalla entre dos entrenadores dentro de un fichero main.py, cuyo esqueleto se proporciona junto con esta práctica. Cada batalla consta de varios combates entre dos Pokémon de los entrenadores. Los entrenadores podrán seleccionar un nuevo Pokemon si su Pokémon se debilita, y la batalla continuará hasta que uno de los entrenadores se quede sin Pokemon. Para implementar este punto, se incluyen dos ficheros de prueba denominados battle0.txt y battle1.txt. El fichero main.py contiene la funcionalidad básica para abrir los ficheros y leer su contenido[5].

[3] Excepto para los atributos name y total_hp para los que solo se definirá el getter.

[4] Ejemplo de salida Bulbasaur (Grass) Stats: Level: 55, ATT: 67, DEF: 29, AGI: 32, HP: 45/55 (donde el 45/55 refleja que el Pokemon ha sufrido 10 puntos de daño respecto a sus puntos de vida totales).

[5] Desde la consola: python main.py battle0.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 battle0.txt

La implementación de la batalla será cómo sigue:

a. Inicio de la batalla. Cada entrenador selecciona su Pokémon inicial, utilizando la función select_first_pokemon(). A continuación, se imprimirá un mensaje indicativo, mostrando los nombres de los entrenadores y los nombres de los Pokémon iniciales que han seleccionado (en adelante, los nombres p1 y p2 son los Pokémon actualmente activos del Entrenador1 y Entrenador2):

==========
Battle between: {trainer1.name} vs {trainer2.name} begins!
{trainer1.name} chooses {p1.name}
{trainer2.name} chooses {p2.name}
==========

donde los valores entre {} indican los valores de los atributos del Pokémon o Entrenador.

b. Inicio del combate entre dos Pokémon. El combate puede durar una o más rondas hasta que uno de los dos Pokémon acaba debilitado. Cada ronda se compone de dos acciones de ataque, una por parte de cada Pokémon. Al inicio de cada ronda se imprimirá un mensaje con la información de los dos Pokemon que se encuentran combatiendo, con el siguiente formato:

┌────────── Round X ──────────┐
Fighter 1: {p1.name} ({p1.pokemon_type}) Stats: Level: {p1.level}, ATT: {p1.strength}, DEF: {p1.defense}, AGI: {p1.agility}, HP: {p1.hp}/{p1.total_hp}.

Figther 2: {p2.name} ({p2.pokemon_type}) Stats: Level: {p2.level}, ATT: {p2.strength}, DEF: {p2.defense}, AGI: {p2.agility}, HP: {p2.hp}/{p2.total_hp}.

Actions

donde X es el número de Ronda, comenzando en 1 y los valores entre {} indican los valores de los atributos del Pokémon.

A continuación, el Pokemon con mayor agilidad es el primero de los dos en atacar (en caso de empate, se priorizará el del primer entrenador). En las rondas impares, los Pokemon usan su ataque específico de clase (p.e. un Pokemon de agua usa su ataque water_attack), mientras que en las rondas pares usan el basic_attack. Con cada ataque se imprimirá un mensaje con la información esencial:

- {attacker.name} uses a {type_attack} on {defender.name}! (Damage: -{damage} HP: {defender.hp})

donde attacker es la instancia del Pokémon que está ataca, defender la instancia del Pokemon que está recibiendo el ataque, y {type_attack} se sustituirá por el string 'basic_attack' si el Pokémon realiza un ataque básico, o por 'water_attack', 'grass_attack', o 'fire_attack' si es un ataque específico de clase.

Además, en el caso de los Pokemon de tipo Planta, y sólo después de llevar a cabo un ataque específico de clase, se curarán con la función heal(), y se imprimirá el siguiente mensaje:

- {attacker.name} is healing! (Healing: +{healing} HP: {attacker.hp})

De manera similar, en el caso de los Pokemon tipo fuego, después de cada ataque específico de clase, llevan a cabo un segundo golpe, siempre que el oponente no esté debilitado, usando la función embers():

- {attacker.name} uses embers on {defender.name}! (Damage: -{damage} HP: {defender.hp})

Si tras un ataque el defensor se debilita[6], se imprimirá el mensaje:

{defender.name} is debilitated

En el momento en que un Pokémon se debilite, se retira y por tanto no puede atacar. Esto puede ocurrir incluso al principio de una ronda, si el ataque de un Pokémon debilita al oponente, éste no realizará su ataque y el combate entre ambos se da por finalizado.

c. Selección de nuevo Pokémon. Si un Pokémon se debilita, su entrenador tiene que seleccionar un nuevo Pokémon, utilizando la función select_next_pokemon(). En caso de que no tenga más Pokemon sin debilitar, se declara al otro entrenador como el ganador de la batalla y se da la ejecución por finalizada imprimiendo el siguiente mensaje.

```
==============================
End of the Battle: {winner.name} wins!
==============================
```

En otro caso, se imprimirá por pantalla el siguiente mensaje con el Pokemon seleccionado (selected_pokemon) y se vuelve a ejecutar el algoritmo desde el paso b:

{trainer.name} chooses {selected_pokemon.name}

Al finalizar la batalla entre ambos entrenadores, se mostrarán varias estadísticas, para lo cual **se utilizará obligatoriamente la librería *pandas***. Se muestra un ejemplo dentro del fichero pandas_example.py, incluido junto con los materiales de esta práctica. No se aceptarán cálculos estadísticos realizados mediante implementaciones ad-hoc que empleen diccionarios u otras estructuras de datos. En concreto deberá mostrarse la media y la desviación estándar para: (1) el daño promedio causado por cada Pokémon individualmente, (2) el daño promedio causado por los Pokémon de cada tipo (agua, fuego, planta), (3) el daño promedio que cada tipo de Pokémon inflige a cada uno de los otros tipos, (4) la curación promedia realizada por cada Pokémon, y (5) la curación promedia realizada por los Pokémon de cada tipo.

Entrega

Se entregará un archivo zip con (exclusivamente) los archivos de código fuente—pokemon.py, trainer.py y main.py—y un manual de usuario (en formato pdf) que incluirá: (1) el mecanismo de ejecución del programa y (2) *una concisa y explicativa* descripción de las fases de desarrollo realizadas. **La memoria no debe exceder las 2 páginas** (tipo de letra Calibri, Arial o Times New Roman con un tamaño mínimo de 11 puntos). La falta de cada apartado penalizará un 10% sobre la nota final. En cada archivo del código fuente y en la primera página del pdf se indicará el nombre y correo electrónico (UDC) de los miembros del grupo de prácticas.

El código de cada clase se documentará con *docstrings* con el formato del ejemplo del ANEXO III. La falta de esta documentación se penalizará con un 10% sobre la nota final.

La práctica debe respetar los principios básicos de orientación a objetos vistos en clase (herencia, abstracción, encapsulación, etc.). Si no se cumplen, la práctica será suspendida automáticamente.

Fecha límite de entrega: viernes, 8 de marzo de 2024 a las 20:00.

Dónde se entrega: en el apartado de Prácticas de Moodle.

Quién entrega: Un sólo miembro de la pareja.

[6] Sus puntos de salud actuales son cero.

ANEXO I - Especificación de los métodos de las subclases Pokemon

WaterPokemon

check_surge_activation()-> bool

Devuelve True si el atributo hp del Pokemon es inferior a total_hp / 2, y False en otro caso.

water_attack(p: Pokemon) -> int

Funcionamiento: Invoca a la función check_surge_activation(), y si devuelve True, establece el atributo surge_mode a True; en caso contrario le asigna False. Además, disminuye la vida del oponente en *n* unidades de daño, calculadas como máximo(1, (factor*$strength_{atacante}$) - $defense_p$), donde factor es igual a 1.5 contra Pokémon de tipo FirePokemon, 1 contra Pokémon de tipo WaterPokemon y 0.5 contra Pokémon de tipo GrassPokemon. El resultado se redondeará siempre al número entero inferior. Al atributo factor se le sumará un 0.1 adicional si el valor del atributo surge_mode es True.

Devuelve: El número de unidades de daño causadas.

Postcondición: La vida del oponente p no debe ser nunca inferior a cero.

effectiveness(p: Pokemon) -> int

Devuelve 1 si p es tipo FirePokemon, 0 si es tipo WaterPokemon y –1 si es tipo GrassPokemon.

GrassPokemon

grass_attack(p: Pokemon) -> int

Funcionamiento: Disminuye la vida del oponente en *n* unidades de daño, calculadas como máximo(1, (factor*$strength_{atacante}$) – $defense_p$), donde el factor es 1.5 contra Pokemon de tipo WaterPokemon, 1 contra Pokemon de tipo GrassPokemon y 0.5 contra Pokemon de tipo FirePokemon. El resultado se redondeará siempre al número entero inferior. A continuación, devuelve el número de unidades de daño causadas al oponente p.

Devuelve: El número de unidades de daño causadas.

Postcondición: La vida del oponente p no debe ser nunca inferior a cero.

heal() -> int

Funcionamiento: Cura al Pokemon con un máximo de *n* unidades, calculadas como el producto del valor del atributo healing y su vida actual, i.e., healing*hp. El resultado se redondeará siempre al entero inferior.

Devuelve: El número de unidades de puntos de vida efectivas en las que se ha curado el objetivo.

Postcondición: Una vez curado, el total de puntos de vida no podrá superar los puntos de vida originales del Pokemon cuando fue instanciado, es decir, no es posible que el Pokemon se cure por encima de sus puntos de vida máximos.

effectiveness(p: Pokemon): int

Devuelve 1 si p es tipo WaterPokemon, 0 si es GrassPokemon y –1 si es FirePokemon.

FirePokemon

fire_attack(p: Pokemon)-> int

Funcionamiento: Disminuye la vida del oponente en *n* unidades de daño, calculadas como máximo(1, (factor*$strength_{atacante}$) – $defense_p$), donde el factor de daño se calcula es 1.5 contra Pokemon de GrassPokemon, 1 contra Pokemon de tipo FirePokemon y 0.5 contra Pokemon de tipo WaterPokemon. El resultado se redondeará siempre al entero inferior. A continuación, devuelve el número de unidades de daño causadas al oponente p.

Devuelve: El número de unidades de daño causadas.

Postcondición: La vida del oponente p no debe ser nunca inferior a cero.

embers(p: Pokemon) -> int

Funcionamiento: Disminuye la vida del oponente en strength*temperature unidades de daño, redondeando siempre al entero más bajo.

Devuelve: El número de unidades de daño causadas.

Postcondición: La vida del oponente p no debe ser nunca inferior a cero.

effectiveness(p: Pokemon) -> int

Devuelve 1 si p es tipo GrassPokemon, 0 si es FirePokemon y –1 si es WaterPokemon.

ANEXO II - Especificación de los métodos de la clase Trainer.

all_debilitated(p: Pokemon) -> bool

Devuelve: True si para todos los Pokémon del entrenador su atributo hp vale cero, False en caso contrario.

select_first_pokemon()-> Pokemon

Devuelve el primer Pokemon del atributo pokemon que no está debilitado, o None en caso de que el entrenador no disponga de Pokemon o estén todos debilitados.

select_next_pokemon(p: Pokemon) -> Pokemon

Precondición: Al menos un Pokemon del Entrenador no está debilitado.

Funcionamiento: Selecciona el Pokemon no debilitado del Entrenador que mejor pueda hacer frente al Pokemon oponente p. Para ello calculará la efectividad con la que cada uno de sus Pokemon puede hacer frente al objetivo usando la función effectiveness(), implementada en la jerarquía de clases Pokemon. Entre aquellos Pokemon con un igual valor de effectiveness, se dará prioridad al Pokemon que sea de mayor nivel.

Devuelve: El Pokemon seleccionado.

ANEXO III – Ejemplo de docstrings

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.

        Parameters
        ----------
        param1 : tipo
             Descripción.

        Returns
        --------
        str
            Resultado de...
        """
```

Docstrings para funciones Python

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

    Parameters
    ----------
    param1 : tipo
        Descripción.

    Returns
    -------
    tipo
Descripción.
    """
```