Informe de Evidencias – Proyecto PyMon Battle

Autores:

- Brandon Camilo Londoño
- Maria Paulina Páez
- Estephany Ruales

Introducción

Este proyecto tiene como objetivo simular una batalla Pokémon por turnos, en la cual un jugador se enfrenta a la "máquina". Para lograrlo, se aplicó el concepto de **Programación Orientada a Objetos (POO)** y sus pilares como abstracción, encapsulamiento, herencia y polimorfismo.

Se trabajó en la separación del código en distintas partes: **Vista**, **Modelo**, **Controlador** y el archivo principal main.py. Esto nos permitió mantener el programa ordenado y fácil de entender.

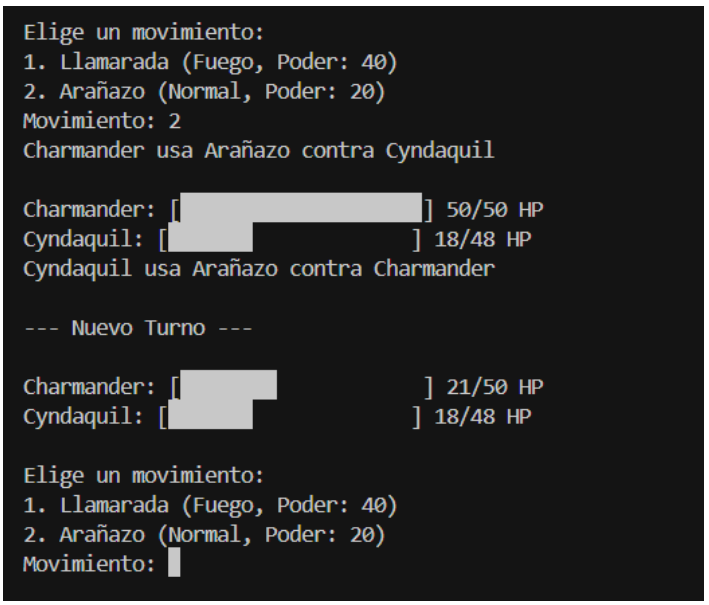
Descripción del Proyecto

El simulador permite que el jugador elija uno de tres Pokémon iniciales: Charmander, Bulbasaur o Squirtle.
La "máquina" selecciona su Pokémon de forma aleatoria. Luego, ambos combaten por turnos hasta que uno de los dos pierda todos sus puntos de vida.

Cada Pokémon cuenta con movimientos propios y tiene ventajas o desventajas según su tipo: **Fuego, Agua o Planta**. Por ejemplo:

- Fuego es fuerte contra Planta, pero débil contra Agua.
- Agua es fuerte contra Fuego, pero débil contra Planta.
- Planta es fuerte contra Agua, pero débil contra Fuego.

El juego también muestra la vida restante de cada Pokémon durante la batalla, lo que hace más visual la experiencia.



Aplicación de los Pilares de la POO

En este proyecto se aplicaron los cuatro pilares fundamentales de la POO:

1. Abstracción

Se identificaron los elementos esenciales para el juego y se representaron mediante clases, ocultando detalles innecesarios.

Por ejemplo:

- La clase Pokemon define lo más importante que debe tener cualquier Pokémon: nombre, tipo, vida, ataque y defensa.
- Esta clase sirve como base para Pokémon específicos como PokemonFuego, PokemonAgua y PokemonPlanta.

Esto permite trabajar de forma clara con conceptos sin preocuparnos por cómo están implementados internamente.

2. Encapsulamiento

La información de cada Pokémon está protegida mediante **atributos privados** (__atributo).

De esta forma:

- Solo se puede acceder a los datos mediante **métodos públicos** como get_vida(), set_vida(), get_nombre(), etc.

- Se evita modificar directamente la vida, ataque o defensa desde fuera de la clase.

```python
class Pokemon(ABC):
    def __init__(self, nombre: str, tipo: str, vida: int, ataque: int, defensa: int):
        self.__nombre = nombre
        self.__tipo = tipo
        self.__vida = vida
        self.__vida_max = vida
        self.__ataque = ataque
        self.__defensa = defensa

    def get_nombre(self):
        return self.__nombre

    def get_tipo(self):
        return self.  tipo
```

3. Herencia

Se utilizó herencia para evitar repetir código y crear relaciones entre clases.

- La clase **base** Pokemon contiene atributos y métodos generales para cualquier Pokémon.
- Las clases hijas (PokemonFuego, PokemonAgua, PokemonPlanta) heredan estas características y añaden comportamientos propios, como movimientos específicos y cálculos de daño según el tipo.

Esto facilita extender el programa agregando nuevos tipos de Pokémon sin modificar la estructura existente.

```python
> class Pokemon(ABC):···

> class Movimiento:···

> class PokemonFuego(Pokemon):···

> class PokemonAgua(Pokemon):···

> class PokemonPlanta(Pokemon):···
```

---

4. Polimorfismo

Cada tipo de Pokémon tiene su propia manera de **atacar**, pero todos comparten el mismo método atacar().
Gracias al **polimorfismo**, no es necesario preocuparse por el tipo específico de Pokémon durante la batalla, ya que el programa puede llamar al método atacar() de manera general, y cada clase se encargará de ejecutar su versión particular de la acción.

- Esto permite que el código sea más flexible y escalable, facilitando la incorporación de nuevos tipos de Pokémon sin necesidad de modificar la estructura principal del programa.

```python
class PokemonFuego(Pokemon):
    def __init__(self, nombre: str, vida: int, ataque: int, defensa: int):
        super().__init__(nombre, "Fuego", vida, ataque, defensa)
        self.movimientos = [
            Movimiento("Llamarada", "Fuego", 40),
            Movimiento("Arañazo", "Normal", 20)
        ]

    def atacar(self, enemigo, movimiento):
        base = movimiento.get_poder() + self.get_ataque() - enemigo.get_defensa()
        if movimiento.get_tipo() == "Fuego" and enemigo.get_tipo() == "Planta":
            base *= 2
        elif movimiento.get_tipo() == "Fuego" and enemigo.get_tipo() == "Agua":
            base *= 0.5
        enemigo.set_vida(enemigo.get_vida() - max(1, int(base)))
        return f"{self.get_nombre()} usa {movimiento.get_nombre()} contra {enemigo.get_nombre()}"
```

Estructura del Código

El proyecto se dividió en los siguientes archivos:

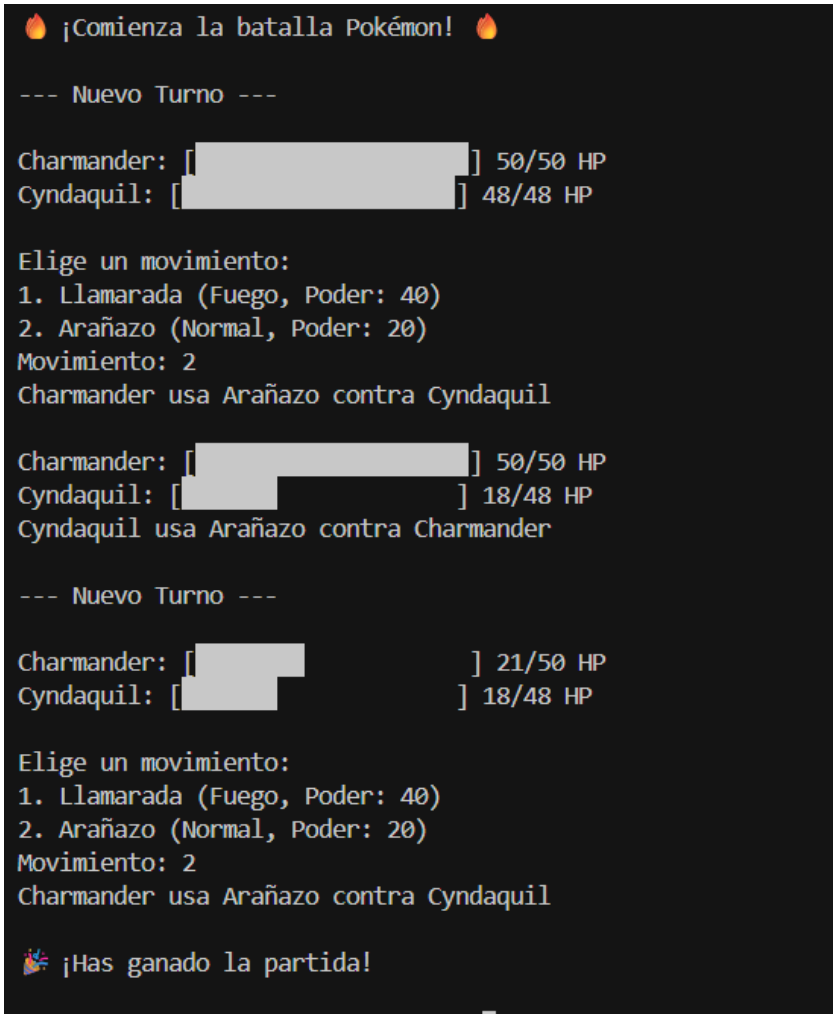
1. **Vista (vista.py)**
    - Se encarga de mostrar mensajes en pantalla y pedir opciones al jugador.
    - Incluye pequeñas pausas para que el juego sea más dinámico.
2. **Modelo (modelo.py)**
    - Contiene la lógica principal del juego.
    - Incluye las clases:
        - Pokemon (clase base).
        - PokemonFuego, PokemonAgua y PokemonPlanta (clases hijas).
        - Movimiento para los ataques.
        - Batalla para gestionar el combate.
        - Entrenador para manejar los equipos de Pokémon.

3. **Controlador (controlador.py)**
    - Coordina la interacción entre la vista y el modelo.
    - Se encarga de preparar los equipos, manejar turnos y definir el flujo de la partida.
4. **Archivo Principal (main.py)**
    - Inicia el juego y permite al jugador jugar varias partidas seguidas.

Evidencias del Funcionamiento

Durante las pruebas se observó que:

- El jugador puede elegir su Pokémon de inicio.
- La máquina selecciona un Pokémon de manera aleatoria.
- Los turnos se alternan correctamente.
- Los ataques muestran mensajes claros y actualizan la vida de los Pokémon.
- Cuando uno de los Pokémon queda sin vida, el juego anuncia el ganador.
- Existe la opción de jugar varias veces sin reiniciar el programa.



Conclusiones

Este proyecto nos permitió aplicar de manera práctica los principios de la Programación Orientada a Objetos (POO) a través del desarrollo de un simulador de batallas Pokémon. La abstracción nos ayudó a identificar los elementos esenciales del juego, como los Pokémon, sus movimientos y las batallas, creando una estructura clara y flexible. El encapsulamiento protegió la información interna de los objetos, evitando modificaciones incorrectas y asegurando la estabilidad del programa. Gracias a la herencia, pudimos reutilizar código y organizar mejor las clases, estableciendo una base común para todos los Pokémon. Finalmente, el polimorfismo brindó flexibilidad al permitir que diferentes tipos de Pokémon ejecutaran el mismo método atacar() con resultados distintos según su tipo. Como resultado, obtuvimos un programa modular y escalable que puede ser ampliado en el futuro con nuevas funciones como evolución, más movimientos y una interfaz gráfica.