Trabajo Práctico nº1

Algoritmos y Programación III

PokeBatalla

Penedo Nicolas 109362

Juan Cruz Robledo Puch..... 106164

Franco Singh..... 108327

Fecha de entrega: 4 / 11 /2023

Introducción:

Como lo aclara el enunciado del TP, El juego es una partida entre 2 jugadores que se enfrentan por turnos a turno hasta que uno alcance la victoria derrotando los pokemones del rival o que el mismo se rinda. Cada uno tiene un máximo de 6 Pokemones y hasta 4 Items. Cada Pokémon tiene un estado, tipo y demás estadísticas. Son duelos 1 a 1 en los que pueden atacarse mediante habilidades, utilizar ítems, posicionar otro pokémon sobre el campo o rendirse.

Hipótesis y Supuestos:

Como se mencionó en clase, los pokemones e ítems iniciales por el momento, son totalmente elegidos arbitrariamente mediante “Generador”. Sin embargo, especulamos y preparamos el programa para poder recibir la información de pokémons e ítems de cada uno de los jugadores mediante JSON, CSV, etc. Al ser un juego con tantas características y partes, tuvimos que tomar distintos supuestos:

- Todos los pokemones empiezan sin estado.
- Si el jugador utiliza una poción que no se puede usar, este pierde el turno.
- el juego al iniciar se le asigna un clima ‘sin clima’, el cual no afecta a los pokemones

Diseño:

Durante el diseño procuramos respetar los pilares de la programación orientada a objetos y programación funcional. “Juego” da inicio y “Generador” se encarga de crear Pokemones e items a usar por el Jugador. El propio juego determina y maneja los turnos durante toda la partida. Nos pareció interesante que Jugador sea el dueño de los Pokemones e Items y así tener control absoluto sobre ellos. Pokemon conoce sus propios atributos y su tipo. Esta es una clase abstracta que implementa un método que utilizan sus clases hijas para saber la efectividad contra otros tipos. Tuvimos un debate sobre qué hacer con las habilidades y finalmente decidimos que cada Pokémon conozca las suyas ya que cada uno tiene diferentes habilidades.

Priorizamos respetar los principios SOLID, básicamente el principio de abstracción y responsabilidades únicas son el tronco del diseño del juego.

Para los Ítem se decidió crear una clase abstracta Items para hacer uso de herencia, ya que los diferentes objetos que plantea el enunciado comparten un comportamiento con el que afectan al estado de un pokémon, pudiendo implementar polimorfismo con el método aplicarItem(). También se planteó lo mismo para el caso de los ítems Modificadores

Para las habilidades también se utilizó Herencia con Polimorfismo en el método usarEnPokemon()

Para los tipos se implementó un doubleDispatch con un diccionario (HashMap) para cada uno de los tipos, en donde se guarda cada uno de los multiplicadores de la tabla y sus respectivas relaciones

Para los estados se decidió hacer un enum, ya que no era viable hacer herencia y polimorfismo como en los otros casos ya que cada “entidad” se comporta de manera diferente sin tener alguna relación más que cumplir con “Es un”. Para el caso de un pokémon sin un estado alterado(ósea normal), el estado del pokémon quede en null, y de querer saber su estado, se mostrará como “NORMAL”

Esta clase existe para hacer de mediador entre los ataques de los pokemones y ver como afecta los estados de los mismos en los turnos. Se pensó así para poder encapsular el comportamiento y la comunicación entre habilidades y pokémons

Para la parte de la Persistencia se crearon objetos Parcer que se encargan de recopilar los datos de los archivos .JSON e integrarlos en el juego

Patrones de Diseño:
Para poder usar la lógica de las habilidades con los pokemon, utilizamos el patron Visitor para usarlo desde un Campo, que conoce a los pokemones e interactúa con una habilidad.
También se implementó el uso de FactoryMethod para la creación de los tipos.
Se usó MVC para el manejo del muestreo de los datos de salida

Conclusión:

En conclusión, utilizar programación orientada a objetos y programación funcional nos ayuda a construir una solución informática consistente, escalable y fuerte. Poder pensar en delegar y ordenar las responsabilidades de cada clase ayuda a un código más ordenado, limpio y menos repetitivo al usar polimorfismo por ejemplo para escalar un producto a algo mucho más grande de lo esperado inicialmente.