DEFENSA 09/05/2024 ENTREGA 21/05/2024

NASHE

PROGRAMACIÓN MODULAR ORIENTADA A OBJETOS

POKEMON

Iker Fernández Molano
María Fernández Pajares
Urko Horas Laguna
Eneko Rodríguez García

ÍNDICE:

INTRODUCCIÓN:

Nuestro juego, y por lo tanto nuestro proyecto, ha consistido en una batalla Pokémon entre dos jugadores. Para jugar, cada jugador, después de introducir su nombre, escogerá 3 Pokémon entre varios de diferentes tipos (agua, planta y fuego) con los que tendrá que luchar contra los 3 Pokémon que haya elegido el oponente hasta que uno de los dos tenga todos sus Pokémon debilitados.
El juego funciona por turnos, como en cualquier juego de Pokémon, y en cada turno, el jugador tiene la opción de atacar, eligiendo entre 4 ataques que tenga el Pokémon que está luchando, la opción de cambiar al Pokémon que está luchando por otro Pokémon que no esté debilitado, y por último, la opción de curar al Pokémon que está luchando.

Una vez la vida de un Pokémon llega a 0, este se debilita, y el jugador deberá elegir uno de los Pokémon restantes que haya elegido al principio para sustituir al debilitado. En el caso de que no tenga ningún Pokémon con vida para cambiar, este jugador habrá perdido la partida.

PLANIFICACIÓN:

Al empezar el proyecto, una vez que decidimos hacerlo sobre un combate Pokémon, lo primero en lo que pensamos fue en el reparto de tareas para la primera entrega. Nos repartimos de modo que Eneko e Iker se encargaron de hacer el diagrama de clases inicial, mientras que Urko y María se encargaron de hacer el diagrama de secuencia inicial.

Tras la primera entrega, era hora de empezar con la preparación del código. A parte del código, tuvimos que corregir ciertos errores e incoherencias que había en los diagramas de clases y secuencia y empezar a realizar las JUnits. Para eso, la planificación fue la siguiente.

- De la corrección del diagrama de clases se ocuparía Iker.
- De la corrección del diagrama de secuencias se ocuparían Urko y María.
- El código lo empezaron a hacer entre Iker y Urko, con la ayuda de ideas que aportaban María y Eneko.
- Por otro lado, María y Eneko se encargaron de empezar a preparar las JUnits a medida que Iker y Urko iban terminando cada clase, para avanzar lo más rápido posible.

Todo esto era la idea inicial que se tenía. Como es lógico, a lo largo de la preparación del proyecto hubo cambios en los planes y cosas que salieron según lo planeado. En ese sentido, cabe destacar que a medida que íbamos avanzando materia en clase, nos pareció interesante ir implementando en nuestro código las cosas nuevas que estudiábamos, además de nuevas ideas que se nos iban ocurriendo. Asimismo, una vez cumplidos los objetivos principales, nos vimos con tiempo e iniciativa para cumplir los objetivos secundarios propuestos en la entrega del DOP, por lo que nos pusimos con ello.

De ahí que el reparto de tareas final ha quedado de la siguiente forma:

- Diagrama de Clases:
    - Iker se encargó del diagrama de clases. (2h)
- Diagrama de Secuencia:
    - Iker elaboró el diagrama de secuencia para el método atacar() (2h)
    - Urko se encargó del diagrama de eliminarPokemon() (2h)
    - Urko, María e Iker trabajaron juntos en el diagrama de empezarPartida() (2h)
    - María realizó el diagrama de imprimir() (30 min)
    - Urko creó el diagrama de listaVacia() (30 min)
    - Eneko se encargó del diagrama de usarCura() (30 min)
- Clases:
    - Iker desarrolló las clases Agua, Fuego, ListaPokemonAElegir, Planta, Pokemon, Tablero y Teclado. (10h)
    - Eneko codificó la clase Ataque. (1h)
    - Urko se encargó de las clases ListaPokemon, Jugador y ListaJugador. (5h)
    - María desarrolló la clase ListaAtaque. (1h)
- JUnits:
    - Eneko se encargó de las pruebas unitarias de Agua, Ataque, Jugador, ListaPokemon y Pokemon. (5h)
    - María desarrolló las pruebas unitarias de Fuego, ListaAtaque, ListaJugador y Planta. (5h)
- Excepciones:
    - Urko implementó las excepciones CambiarDePokemonException y UsarCuraException. (2h)

DISEÑO: DIAGRAMA DE CLASES

- En un principio el diseño no tenía herencia, en la clase Pokemon, ya que los distintos tipos de pokemon era un objetivo secundario. Cuando abordamos este objetivo, vimos conveniente hacerlo de esta manera, porque a largo plazo, si seguiríamos actualizando el juego, sería muy sencillo añadir más tipos de **Pokemon**, sólo habría que crear las nuevas clases y definir cuándo sería supereficaz() o pocoeficaz().
- Además, tampoco se hacía uso de ninguna excepción, porque también era una herramienta que hemos acabado utilizando en el objetivo secundario, de cambiar los pokemons en batalla y usar curas.
    - **CambiarDePokemonException**, se lanza desde atacar() en Pokemon cuando se pulsa <<5>>.
    - **UsarCuraException**, en cambio, se lanza cuando se pulsa <<6>> en el Teclado.
- Por otro lado, se ha añadido en el diagrama de clases final, la clase **ListaPokemonAElegir**, que son 12 pokemons que se dan para elegir al inicio de la partida, los pokemons con los que jugarás la partida.
- Se han añadido distintos subprogramas a las clases para poder realizar los objetivos secundarios, así como, cambiarPokemon() de **Tablero** o dismunuirUso() de **Ataque** (Que fue un objetivo que se abordó desde un inicio).
- Asimismo, en la clase **Tablero**, se han creado distintos programas para poder "atomizar" las distintas tareas del juego. Así sería más fácil detectar un posible error en el programa.

DIAGRAMA DE CLASES (Diseño inicial)

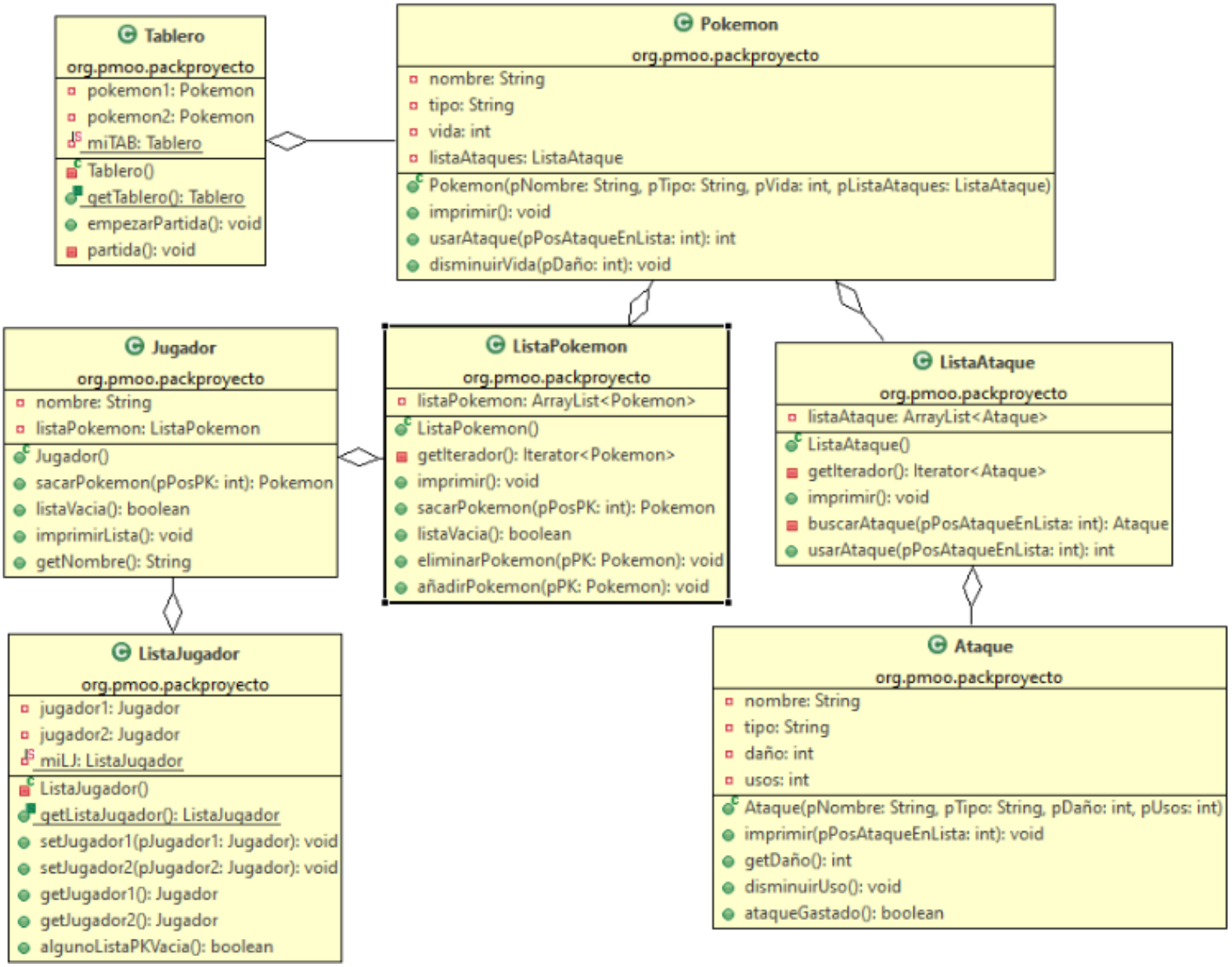


DIAGRAMA DE CLASES (Diseño Final, sin objetivos secundarios)

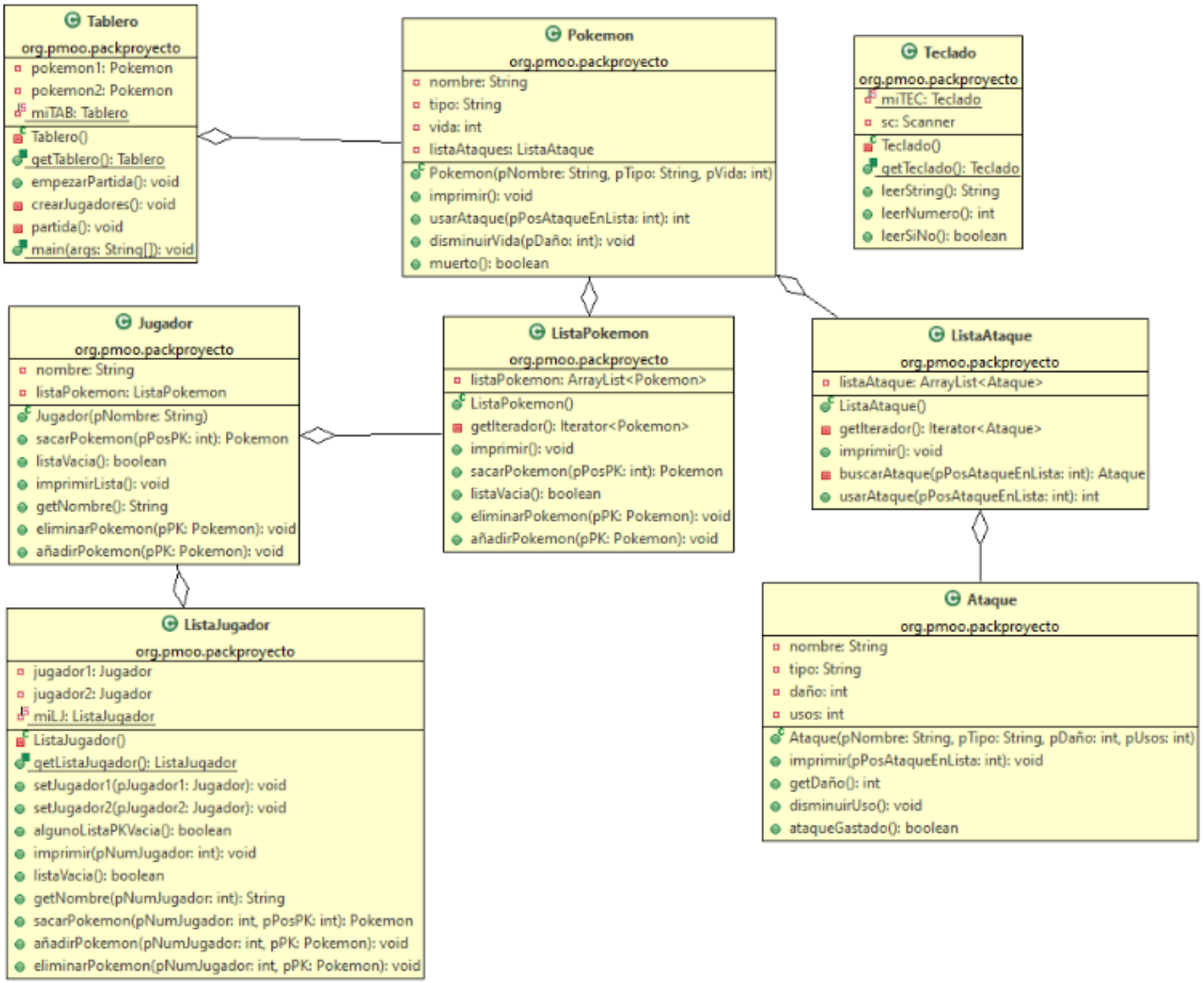
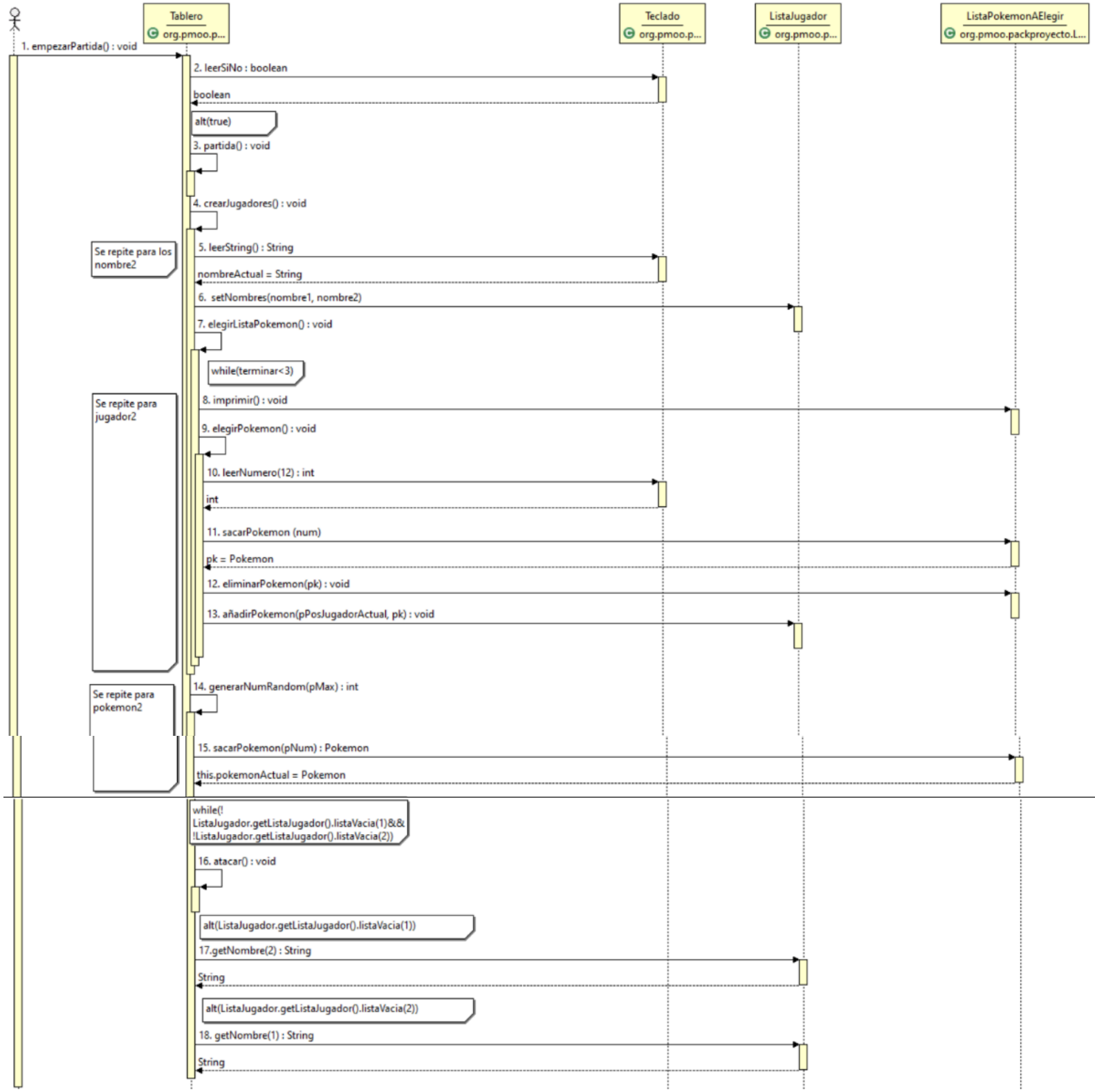


DIAGRAMA DE CLASES (Diseño final)

DISEÑO: DIAGRAMA DE SECUENCIAS

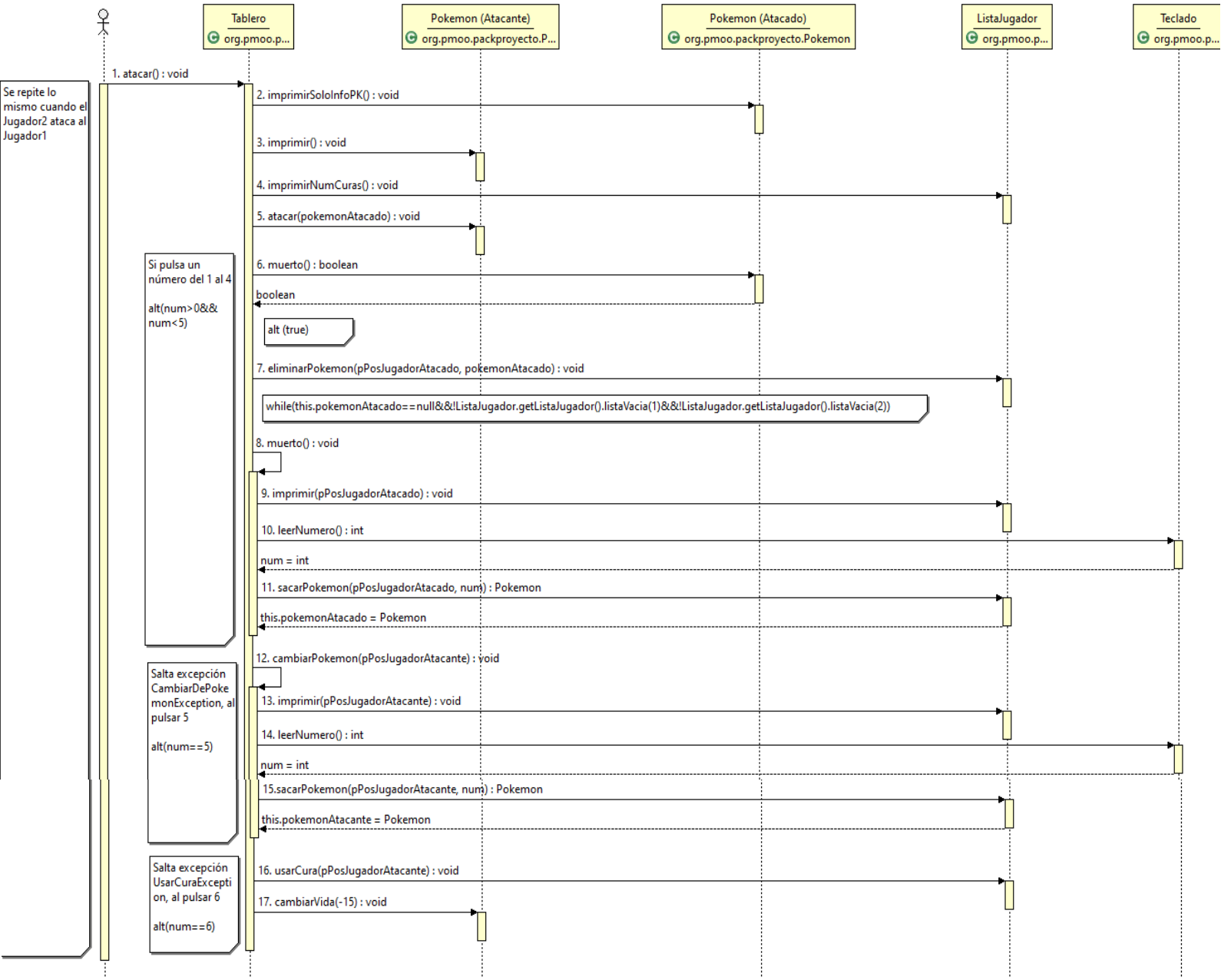
empezarPartida()



LLAMADAS:

1. empezarPartida(): void → Es la llamada que se hace desde el usuario al tablero para empezar la partida.
2. leerSiNo: boolean → Se llama a la clase teclado para pedir al usuario un "si" o un "no" que se devolverá en forma de booleano.
3. partida(): void → Se llama al método partida que está en la propia clase tablero. Este método no devuelve nada, pero se encarga de crear jugadores, pokemons...
4. crearJugadores(): void → Se le pide al usuario el nombre de los jugadores 1 y 2, se establecen y pasan a elegir su lista de pokemons.
5. leerString(): String → Se llama a la clase teclado para pedir al usuario el nombre del jugador 1 y 2.
6. setNombres(nombre1, nombre2): void → Se establecen los nombres devueltos por el anterior método, en las variables jugador1 y jugador2 en la clase ListaJugador.
7. elegirListaPokemon(): void → Desde ListaPokemonAElegir se imprime la lista de pokemons disponibles a elegir y en tablero se llama al método elegirPokemon().
8. imprimir(): void → Se encarga de imprimir la lista de pokemons disponibles a elegir desde la clase ListaPokemonAElegir.
9. elegirPokemon(): void →
10. leerNumero(12): int → Se devuelve el número que se le pide al usuario, siempre y cuando sea menor que el valor que se pasa por parámetro, en este caso un valor constante 12, desde la clase teclado.
11. sacarPokemon(num): Pokemon → Desde ListaPokemonAElegir se devuelve el pokemon que hay en la posición indicada por el parámetro.
12. eliminarPokemon(pk): void → Desde ListaPokemonAElegir se elimina el pokemon indicado por el parámetro.
13. añadirPokemon(pPosJugadorActual, pk): void → Desde ListaJugador se añade a la lista de pokemons del jugador indicado por el parámetro pPosJugadorActual el pokemon indicado por el parámetro pk.
14. generarNumRandom(pMax): int → Se genera un número aleatorio menor que el numero en el parámetro pMáx.
15. sacarPokemon(pNum): Pokemon → Desde ListaPokemonAElegir se devuelve el pokemon que hay en la posición indicada por el parámetro.
16. atacar(): void → Se hace la llamada al método atacar que se explicará en un diagrama de secuencia a parte y no devuelve nada.
17. getNombre(2): String → Devuelve el nombre del jugador 2.
18. getNombre(1): String → Devuelve el nombre del jugador 1.

atacar()


Tablero
org.pmoo.p...
Pokemon (Atacante)
org.pmoo.packproyecto.P...
Pokemon (Atacado)
org.pmoo.packproyecto.Pokemon
ListaJugador
org.pmoo.p...
Teclado
org.pmoo.p...
1. atacar() : void
Se repite lo mismo cuando el Jugador2 ataca al Jugador1
2. imprimirSoloInfoPK() : void
3. imprimir() : void
4. imprimirNumCuras() : void
5. atacar(pokemonAtacado) : void
Si pulsa un número del 1 al 4
alt(num>0&& num<5)
6. muerto() : boolean
boolean
alt (true)
7. eliminarPokemon(pPosJugadorAtacado, pokemonAtacado) : void
while(this.pokemonAtacado==null&&!ListaJugador.getListaJugador().listaVacia(1)&&!ListaJugador.getListaJugador().listaVacia(2))
8. muerto() : void
9. imprimir(pPosJugadorAtacado) : void
10. leerNumero() : int
num = int
11. sacarPokemon(pPosJugadorAtacado, num) : Pokemon
this.pokemonAtacado = Pokemon
12. cambiarPokemon(pPosJugadorAtacante) : void
Salta excepción CambiarDePokemonException, al pulsar 5
alt(num==5)
13. imprimir(pPosJugadorAtacante) : void
14. leerNumero() : int
num = int
15.sacarPokemon(pPosJugadorAtacante, num) : Pokemon
this.pokemonAtacante = Pokemon
Salta excepción UsarCuraException, al pulsar 6
alt(num==6)
16. usarCura(pPosJugadorAtacante) : void
17. cambiarVida(-15) : void

LLAMADAS:

1. atacar(): void → Es la llamada que se hace desde el usuario al tablero para empezar a atacar.
2. imprimirSoloInfoPk(): void → Se llama a la clase Pokemon(Atacado) para imprimir la información del pokémon atacado.
3. imprimir(): void → Se encarga de mostrar por pantalla la información del pokémon atacante elegido y la lista de ataques.
4. imprimirNumCuras(): void → Se llama a la clase ListaJugador para imprimir el número de curas que le cada a cada jugador.
5. atacar(pokemonAtacado): void → Se llama a la clase Pokemon(Atacante) pasándole por parámetro el pokemon que quiere que sea atacado y dependiendo del número que teclee el usuario, hace una función distinta.
6. muerto(): boolean → Llama a la clase Pokemon(Atacado) y devuelve un true si ese pokémon tiene vida 0 y está muerto. Sin embargo, en el caso contrario, devolverá un false.
7. eliminarPokemon(pPosJugadorAtacado, pokemonAtacado) : void → si el método anterior devuelve un true, entonces se llama a la clase ListaJugador para eliminar el pokemon muerto de la lista de pokemons.
8. muerto(): void → mientras que el pokemon atacado sea nulo y las listas de los dos jugadores no sean vacías, se vuelve a llamar al método muerto() desde la clase Tablero.
9. imprimir(pPosJugadorAtacado): void → Desde la clase Tablero, llama a la clase ListaJugador e imprime por pantalla toda la información de la lista de pokemons del jugador atacado.
10. leerNumero(): int → Desde la clase Tablero, llamamos a la clase Teclado para saber que ha tecleado el usuario y dependiendo del número que devuelva el método, habrá tareas distintas.
11. sacarPokemon(pPosJugadorAtacado, num) : Pokemon → Desde la clase Tablero, llamamos a la clase ListaJugador que se encargará de sacar el pokémon del jugador elegido y que está en la posición recibida por parámetro. Además, nos devuelve el pokémon elegido.

Desde el número 8 al 11, sucederán si la condición del while se cumple.

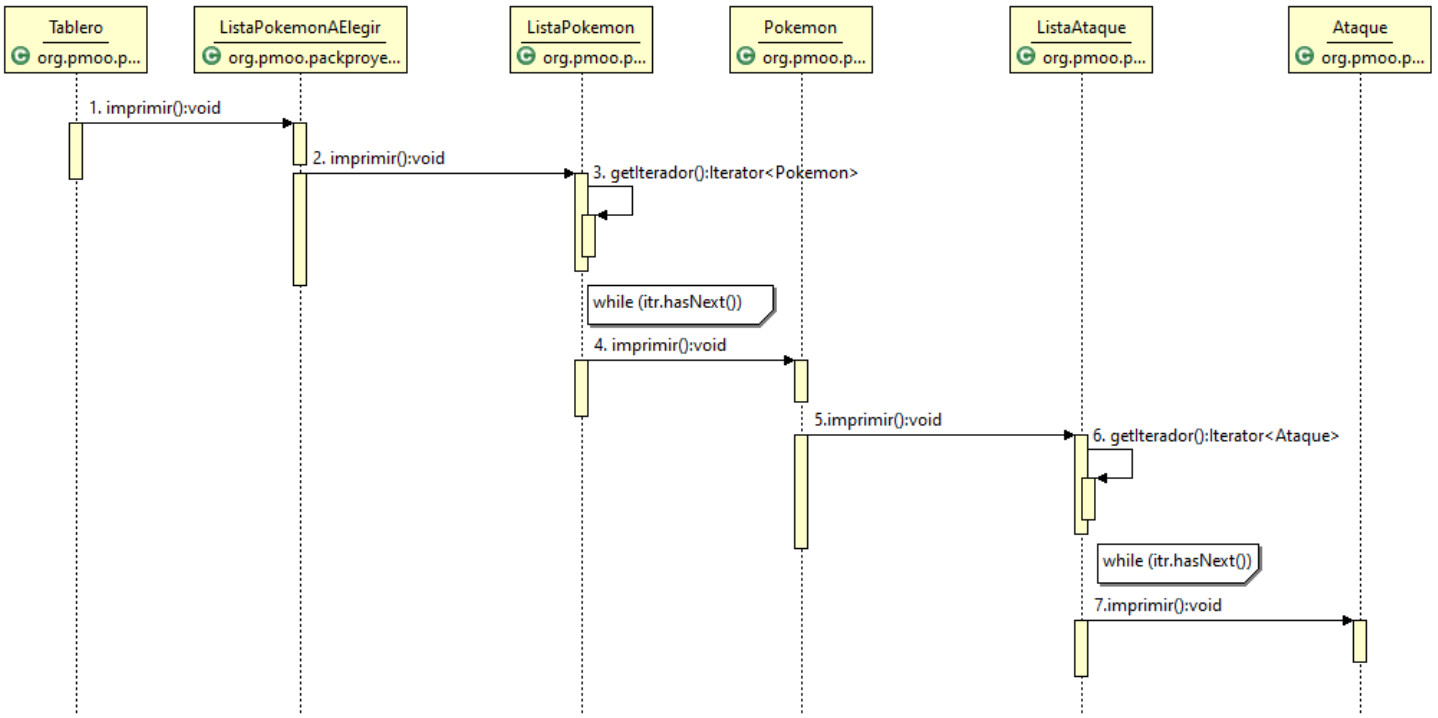
12. cambiarPokemon(pPosJugadorAtacante) : void → En el caso de que no se cumpla la condición del while, se llamará desde el Tablero al método para cambiar el pokémon del jugador elegido.
13. imprimir(pPosJugadorAtacante): void → se llama a la clase ListaJugador que imprimirá la lista de pokémon que quedan.
14. leerNumero() : int → Desde la clase Tablero, llamamos a la clase Teclado para saber que ha tecleado el usuario y dependiendo del número que devuelva el método, habrá tareas distintas.
15. sacarPokemon(pPosJugadorAtacante, num) : Pokemon → Desde la clase Tablero, llamamos a la clase ListaJugador que se encargará de sacar el pokémon del jugador elegido y que está en la posición recibida por parámetro. Además, nos devuelve el pokémon elegido.

Desde el número 12 al 15, salta excepción CambiarDePokemonException, al pulsar el número 5 en el teclado.

16. usarCura(pPosJugadorAtacante) : void → Desde la clase Tablero, se llama a la clase ListaJugador que se encargará de usar la cura para que la vida del pokémon sea restablecida.
17. cambiarVida(-15) : void → Desde la clase Tablero, llamamos a la clase Pokemon para cambiarle la vida al pokemon.

Desde el número 16 al 17, salta excepción UsarCuraException, al pulsar el número 6 en el teclado.

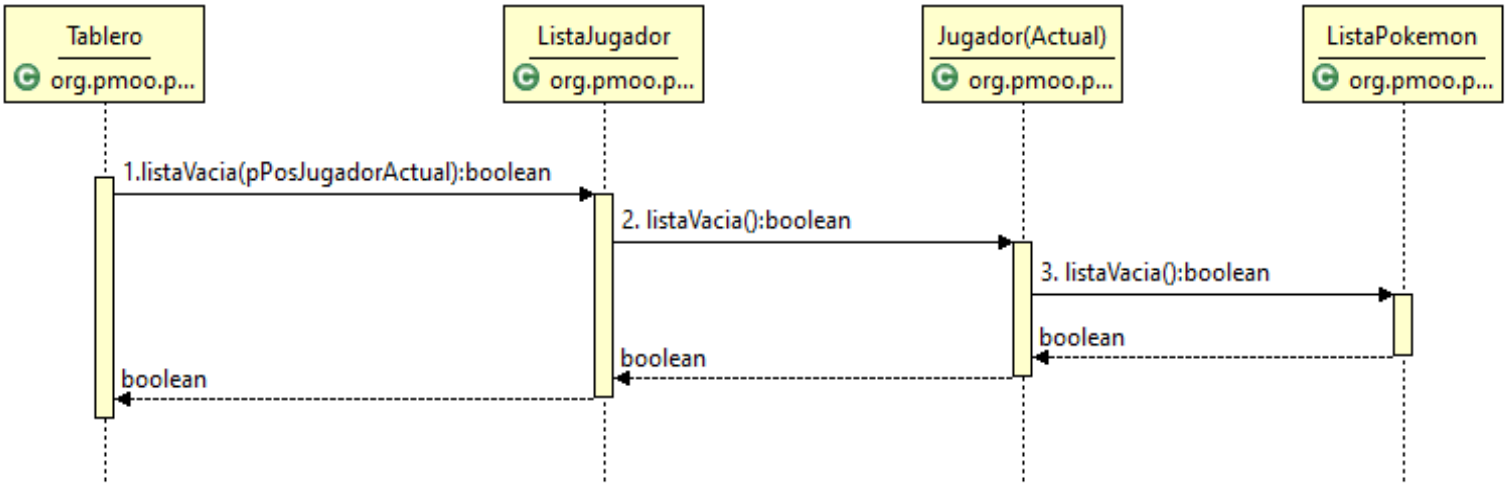
imprimir()



LLAMADAS:

1. imprimir(): void → Desde la clase Tablero, llamamos a la clase ListaPokemonAElegir que se encargará de llamar a otros métodos para imprimir la información de los pokemons.
2. imprimir(): void → Desde la clase ListaPokemonAElegir, se llama a la clase Lista Pokemon para imprimir toda la lista de los pokemons.
3. getIterador():Iterator<Pokemon> → Desde la clase ListaPokemon, se llama al iterador que se encarga de ir imprimiendo uno a uno la lista.
4. imprimir(): void → Mientras que siga habiendo pokemons en la lista, se irá imprimiendo la información de cada pokemon.
5. imprimir():void → Desde la clase Pokemon, se llama a la clase ListaAtaque para imprimir todos los ataques, además de los pokemons que se han imprimido en el método anterior.
6. getIterador():Iterator<Ataque> → Desde la clase ListaAtaque, se llama al iterador que se encarga de ir imprimiendo uno a uno la lista.
7. imprimir(): void → Mientras que siga habiendo ataques en la lista, se irá imprimiendo la información de cada ataque.

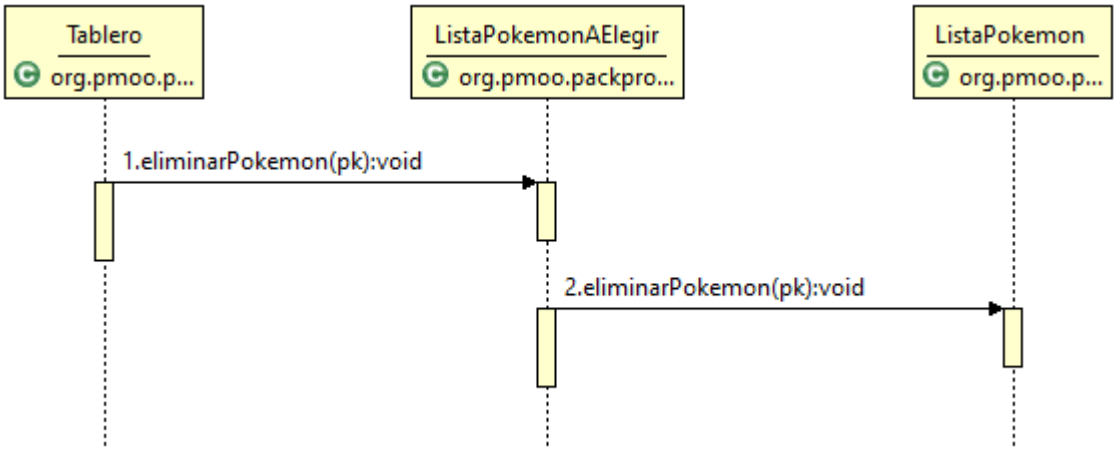
listaVacia()



LLAMADAS:

1. listaVacia(pPosJugadorActual): boolean → Desde la clase Tablero, se llama a la clase ListaJugador que se encarga de comprobar si la lista del jugador que se ha pasado por parámetro es vacía o no.
2. listaVacia(): boolean → Desde la clase ListaJugador, se llama a la clase Jugador que se encarga de comprobar si la lista de pokemons del jugador es vacía o no.
3. listaVacia(): boolean → Desde la clase Jugador, se llama a la clase ListaPokemon que se encarga de comprobar si la lista de pokemons es vacía o no.

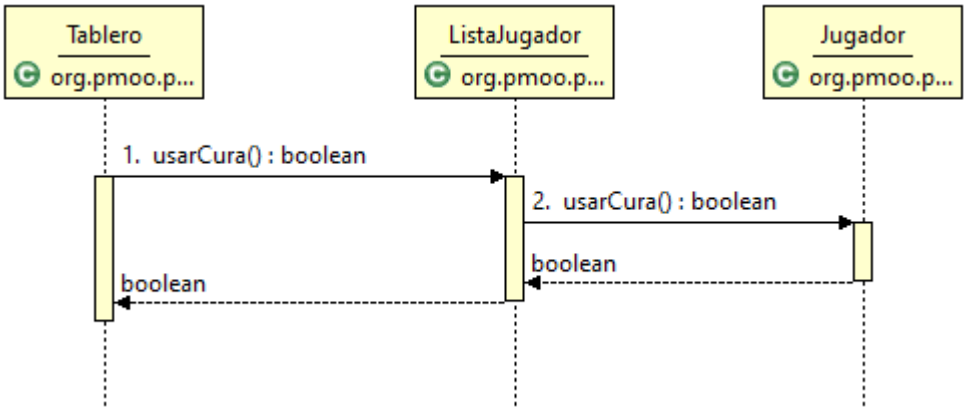
eliminarPokemon()



LLAMADAS:

1. eliminarPokemon(pk): void → Desde tablero se hace la llamada al método eliminarPokemon() de la clase ListaPokemonAElegir en el que simplemente se hace una llamada al método eliminarPokemon() de la clase ListaPokemon.
2. eliminarPokemon(pk): void → Desde ListaPokemonAElegir se hace la llamada al método eliminarPokemon() de la calse ListaPokemon en el que se utiliza la sentencia remove para eliminar el pokemon que se pasa como parámetro.

usarCura()



LLAMADAS:

1. usarCura(pNumJugador):boolean → Desde tablero se hace la llamada al método usarCura() de la clase ListaJugador en el que se va a comprobar si el jugador actual (depende del parámetro) tiene sus curas agotadas o no con el booleano que devuelve, mientras se hace la llamada al método usarCura() de la clase Jugador.
2. usarCura(pNumJugador):boolean → Desde ListaJugador se hace la llamada al método usarCura() de la clase Jugador en el que se comprueba si al jugador le quedan curas o no. En caso afirmativo, se le resta una cura. De lo contrario, se devuelve true, y por tanto no podría usar curas, lo que se imprimiría por pantalla.

CASOS DE PRUEBA: Diseño de las JUnits

Vamos a explicar brevemente los casos de prueba que hemos implementado para comprobar que nuestro proyecto funciona correctamente.

➥ ***Clase Ataque:***

- ⇨ DisminuirUso(): comprobamos que disminuye los usos al ataque elegido.
- ⇨ AtaqueGastado(): comprobamos que funciona correctamente si ese ataque ha gastado ya todos los usos. En caso de que hayan sido usados, ese ataque ya no se podrá usar.

➥ ***Clase Jugador:***

- ⇨ SacarPokemon(): este método puede lanzar una excepción; por lo tanto, tenemos que comprobar si este método hace sus funciones correctamente. Si le pasan por parámetro la posición de un pokémon que no está en la lista, debería saltar la excepción IndexOutOfBoundsException. Sin embargo, si le pasan una posición correcta, debería saltar la excepción.
- ⇨ EliminarPokemon(): este método también lanza una excepción. Al eliminar un pokémon, tenemos que comprobar que llamando al método sacarPokemon(), ese mismo ya no está en la lista. En el caso contrario, deberá saltar la excepción.
- ⇨ AñadirPokemon(): este método también lanza una excepción. Añadimos un pokemon y comprobamos que al llamar al método sacarPokemon(), devuelve el pokemon que hemos añadido. En el caso contrario, si después de añadirlo no lo encuentra en la lista, debería saltar la excepción.

Estos métodos, que están también en la clase ListaJugador y ListaPokemon, hacen una llamada a estos mismos métodos pero en las distintas clases. Por lo tanto, para comprobar que estos métodos funcionan correctamente, comprobamos que el pokémon se ha añadido o eliminado correctamente si saca aquel que está en la posición pasada por parámetro.

➥ ***Clase ListaAtaque:***

- ⇨ UsarAtaque(): al crear un nuevo ataque y añadirlo a la lista, tenemos que comprobar que ese ataque está añadido correctamente. Para ello, comprobamos que funciona si llamando a este método, nos devuelve el valor que esperábamos.
- ⇨ AñadirAtaque(): creamos un nuevo ataque, lo añadimos a la lista y comprobamos que se ha añadido correctamente mediante la llamada a usarAtaque(), nos devuelve el valor esperado.

➥ ***Clase Pokemon:***

- ⇨ CambiarVida(): con este método, lo único que tenemos que comprobar es que al pokemon que le hemos cambiado la vida con el daño pasado por parámetro, su vida no sea cero y no esté muerto.
- ⇨ Muerto(): al contrario que el anterior, tenemos que comprobar que al pokemon que le hemos cambiado la vida, ya que se le pasa por parámetro el daño que se le hace, su vida sea 0 y por lo tanto esté muerto.

➥ ***Clases Agua, Fuego y Planta:***

- ⇨ superEficaz(): este método en cada clase es diferente; por lo tanto, tenemos que comprobar en cada uno de ellos que pokemons tienen un ataque super eficaz y cuáles no.
- ⇨ pocoEficaz(): este método es el contrario al anterior. Por lo tanto, comprobamos qué ataques son poco eficaces.

En las JUnits que hemos implementado también estarían las pruebas de las constructoras de cada método, los getters y setters necesarios de cada clase, y los tests para comprobar que las listas no estaban vacías.

Cabe mencionar también, que Tablero ha sido *testeado* directamente en partida, pues pide datos por medio de Teclado.

EXCEPCIONES

Después de dar las excepciones en las clases magistrales, decidimos añadir algunas cosas a nuestro juego para poder aplicar lo aprendido. Decidimos añadir las opciones de usar cura y de cambiar Pokémon.

Para ello, hemos creado las clases usarCuraException y cambiarPokemonException que se utilizan cuando al pedir un número al jugador para que elija un ataque para lanzar.
Si el jugador introduce un número entre el 1 y el 4, se utilizará el ataque correspondiente, sin embargo si el jugador introduce el número 5 o el número 6, saltará la excepción que corresponda y se tratarán de forma que se utilice una cura o se cambie el Pokémon.

ASPECTOS MÁS DESTACABLES DE LA IMPLEMENTACIÓN

En nuestro programa no hemos utilizado ninguna extensión o librería externa. Sin embargo, cabe mencionar que hemos hecho uso de herramientas ya implementadas en Java como la utilidad Random, que utilizamos en Tablero solamente para que al principio de la partida salga al "campo de batalla" un Pokemon aleatorio entre los 3 que tiene cada jugador en su lista.

Cabe destacar también, que aunque todos los subprogramas que hay implementados en Tablero podrían resumirse en uno único. A la hora de implementar el diseño optamos por intentar separar al máximo cada tarea que realiza el programa, para no sólo facilitar la comprensión de un individuo que nunca haya tocado el código, también para facilitar nuestro trabajo y organización.

Y siguiendo por la misma línea, hemos intentado hacer los programas los más versátiles posibles. Para que si se quiere, por ejemplo, jugar con más Pokemon en cada lista, sólo haya que cambiar un único parámetro del código. Lo mismo ocurriría, si queremos añadir más tipos, solamente habría que crear las nuevas clases hijas de Pokemon.

CONCLUSIONES

En nuestra opinión, hemos hecho un proyecto muy completo. Hemos intentado utilizar todo lo aprendido en clase, a medida que lo íbamos dando. Por ejemplo, cuando dimos la herencia, optamos por hacer uso de esta para los distintos tipos de Pokemon. De esta manera, no sólo nos ayudaría a tener un acabado más organizado y completo, si no que también nos ayudaría a afianzar las distintas herramientas, además, de hacer los laboratorios.

Además, un reflejo del buen trabajo en grupo y coordinación que ha habido es que hemos logrado implementar todos los objetivos secundarios que se propusieron al principio del diseño. Asimismo, ha sido fácil de ir implementandolos mientras hacíamos el código, por lo bien estructurado que estaba el código.

En general, todo el grupo ha tenido buena comunicación entre sí y se han respetado los tiempos para realizar cada tarea.

Con respecto a los problemas que hemos tenido durante la implementación, no han sido muchos e incluso ninguno, ya que teníamos una idea bastante clara desde el principio de cómo podríamos abordar cada tarea.

Durante la defensa y el turno de preguntas, hemos sacado como conclusión lo importante que es que todo el grupo esté bien comunicado, pues todos los miembros son “padres/madres ” del proyecto y deben ser capaces de responder a cualquier pregunta. Y que el trabajo y tiempo que se ha dedicado al proyecto se refleja a la hora de saber cómo y qué responder a las preguntas que son planteadas.