2013

PMOO Conquerors of the League

Mikel Barcina
Mikel Dalmau
Jose Ángel Gumiel
Begoña Mtnez. De Marañón
Jonatan Pérez

ÍNDICE

CONTENIDO

ÍNDICE

CONTENIDO

Introducción

A lo largo de las siguientes páginas vamos a presentar nuestro proyecto. Explicaremos en qué consiste, cómo se desarrolla internamente, qué metodología hemos seguido para organizarnos y trabajar en grupo, qué conocimientos previos hemos necesitado y qué conclusiones hemos obtenido. Además expondremos otros aspectos que nos parecen interesantes de recalcar.

Descripción del juego

Nuestro proyecto está inspirado en el famoso videojuego Pokémon, que tuvo tanta popularidad que llegó a convertirse en una serie de animación televisiva.

Para nuestro proyecto hemos cogido un entrenador, nada más iniciar el juego se le pregunta por el nombre, y posteriormente se le pide que elija una carta. Esa carta será con la que tenga que luchar. A lo largo del camino se encontrará con otros personajes que también portarán una carta. Nuestro protagonista deberá de combatir para hacer que su carta suba niveles y sea más fuerte. En los combates se puede perder, lo que conllevará, como penalización, la pérdida de niveles. En el enfrentamiento también se pierde vida, por eso hay que estar atento para decirle a la carta que recupere vida.

Los personajes que se puede encontrar son de tres tipos:

- Entrenadores: En cada casilla del mapa hay un entrenador. Con estos se puede luchar o no luchar, es el usuario el que decide. Si se enfrenta a ellos y pierde, la carta bajará de nivel. En el caso de que gane la carta subirá de nivel, y el contador del entrenador se aumentará en +1. No podrá ganar al mismo entrenador más de cuatro veces. El motivo es que el protagonista podría pelear infinitas veces con una carta muy inferior a él, hasta tener el nivel suficiente como para pasarse el juego sin esforzarse en absoluto.
- Líderes: El líder es una forma de frenar al protagonista, si no les derrotas al menos una vez, no podrás pasar a la siguiente casilla. Algunos líderes podrán tener más de una carta, pero esta no la usarán para combatir, sino que será utilizada a modo de recompensa. Si te enfrentas a un líder con una carta extra y le ganas, este te la entregará, y tendrás dos cartas. Sigue con el mismo patrón que el entrenador, tienen un contador para que les ganes a ellos más de 4 veces.
- Élite: Se componen de tres entrenadores con tres cartas diferentes. Se encuentran siempre al final de cada nivel. Sólo les puedes ganar una única vez, ya que cuando les ganes pasarás automáticamente al siguiente nivel, y no tendrás posibilidad de retroceder.

Por otro lado, tenemos cuatro tipos de cartas, que son Agua, Fuego, Hoja y Psico. En el caso de que tengamos más de una carta podemos decidir con cual nos queremos enfrentar. El de Fuego tiene ventaja sobre Hoja, el de Hoja tiene ventaja sobre Agua, el de Agua tiene ventaja contra Fuego. El de Psico, por el contrario es neutral.

El juego se compone de 5 niveles, cada nivel tiene 7 casillas, cuando se llega a la séptima y se derrota a la élite pasamos a la primera casilla del siguiente nivel.

Tenemos que comentar que este juego no tiene fin. En un principio se gana cuando se llega a la última casilla del último nivel, pero es recursivo, tenemos la oportunidad de seguir jugando. En tal caso se vuelve a iniciar en el nivel uno con las mismas narraciones, pero con unas cartas a un nivel superior, ya que no sería equitativo volver a enfrentarse contra cartas de nivel 5 cuando el protagonista tiene un nivel 200.

Motivaciones

Antes de empezar con este proyecto teníamos otro en mente, pero no lo vimos viable para esta asignatura, así que buscamos algo diferente.

Este juego puede empezar de una forma muy sencilla, sin embargo, si así lo queremos, lo podemos complicar y añadir nuevas funcionalidades, es muy abierto. Tenemos un mapa, que son casillas, en esas casillas nosotros hemos introducido personajes, cartas y cartas regalo. Sin embargo, si hubiéramos dispuesto de más tiempo podríamos haber añadido objetos, como por ejemplo pociones. Otra de las cosas que podría tener es un método para poder conseguir la carta del rival en el caso de que dispusiéramos de un objeto como un “receptáculo”. O la posibilidad de que la carta tuviera varios ataques contenidos en una “ListaDeAtaques”. Realmente se pueden añadir muchas ideas nuevas sin tener que cambiar apenas la estructura del juego. De hecho, dos días antes de la presentación implementamos la opción de que el jugador dispusiera de más de una carta.

También pensamos que gran parte del código podría ser reutilizable para hacer un juego distinto. El mapa se puede cambiar de tamaño fácilmente, y añadir nuevos lugares y nuevas narraciones no resulta ningún problema. También podemos cambiar a los entrenadores por enemigos en el caso de querer hacer cambios, por poner un ejemplo.

En definitiva, queríamos algo sencillo de hacer, para poder cumplir con nuestros objetivos, pero que la finalización de esos objetivos no supusieran la finalización del juego y del proyecto.

Aspectos técnicos

Durante el desarrollo del juego hemos tenido que recurrir a los conocimientos explicados en clase y a las prácticas de laboratorio. Si no hubiéramos practicado antes de lanzarnos con el proyecto, habría sido un fracaso. Además de esto, también nos ha servido para buscar información, utilizar comandos que hasta ahora desconocíamos y para pensar dónde deben ir los métodos para que el programa funcione lo mejor posible.

Hasta ahora, en los laboratorios, nos han dado los diagramas ya hechos, pero elaborar uno desde cero es otra historia, y como hemos visto, la primera solución dista mucho de la solución final que hoy exponemos.

Objetivos del proyecto

Podemos decir que hemos cumplido la totalidad de los objetivos marcados en el DOP. Los que propusimos en su día son los siguientes:

Objetivos principales:

- ✓ Implementar de forma correcta las Clases y métodos más básicos para que se pueda jugar
- ✓ Tener un juego que sea ejecutable por consola y que admita una interacción por parte del usuario.
- ✓ Diseñar varios niveles y que haya métodos y atributos que puedan jugar un papel importante para romper con la rutina del juego y afecten a la forma de competir en las luchas.

Objetivos artísticos:

- ✓ La creación de un ejecutable que permita ejecutar el programa fuera de eclipse, en la consola de Windows (cmd). Es muy sencillo, basta con crear un archivo .bat en el bloc de notas que contenga el siguiente comando:

  "javaw -Xmx32M -jar nombreArchivo.jar"

  Esto nos parece interesante porque en un principio se nos prohibió el uso de la interfaz gráfica, lo cual es correcto, ya que nos desvía de los objetivos principales, sin embargo con una sola línea de comando nos permite mostrar el juego sin necesidad de tener un IDE instalado, y de este modo poder mostrarlo.
- ✓ Un hilo argumentativo que permita seguir una historia coherente y que se ajuste al juego que proponemos.

Objetivos adicionales:

- ✓ Aprender a diseñar un proyecto desde cero, administrar el tiempo, poner metas a corto plazo y cumplirlas, crear unos diagramas correctamente que favorezcan al desarrollo de nuestra aplicación.
- ✓ Tener el juego terminado completamente, implementado y con las JUnits correspondientes.
- ✓ Variedad, hemos utilizado MAE, TAD, ArrayList, Herencia, Excepciones… Así como comandos no utilizados en clase, break, thread.sleep, scanner…

Podemos decir que estamos contentos con el trabajo realizado, y que incluso hemos tenido tiempo para añadir algunos aspectos que no estaban dentro de lo planeado y conseguir que todo funcione correctamente.

Planificación realizada

En este proyecto hemos invertido muchas horas, y la verdad es que consideramos que se ha notado en el resultado final y que ha merecido la pena.

Durante las semanas que hemos estado trabajando en el proyecto hemos quedado cómo mínimo dos días, Jueves y Viernes.

Los jueves hemos trabajado en el laboratorio de programación, y los viernes nos hemos quedado a partir de las 15h. en el aula de ordenadores de la facultad. Juntando estos dos días podemos decir que hemos dedicado como mínimo entre 5 y 7 horas semanales. Algunas semanas hemos estado más horas, y hemos llegado a quedar también algún Lunes. Además de esto, tenemos que añadir el trabajo individual que cada uno ha realizado por su cuenta en casa.

No hemos sido muy estrictos a la hora de quedar entre nosotros, hay días en los que miembros del grupo no se han reunido. Pero tampoco fue necesario. Sabíamos que iban a cumplir, que realizarían su trabajo y así fue. No hemos dado demasiada importancia a las horas de reunión, sino a las tareas asignadas y a su realización.

El no estar reunidos presencialmente no significa no estar en contacto. Hemos trabajado mediante dos plataformas, por un lado, para las discusiones e intercambio de ideas, hemos utilizado un grupo de Whatsapp en el que estábamos los cinco miembros del grupo. Para compartir el trabajo, y tener la última versión del proyecto accesible, teníamos una carpeta compartida en DropBox. Todos podíamos subir contenidos y trabajar sobre la última actualización. Como había cosas que íbamos borrando o cambiando, cada día teníamos una copia del workspace actualizado. Es decir, tenemos varios archivos comprimidos de fechas distintas, así si quitamos algo que luego necesitamos o empeoramos el código, sabemos que tenemos un respaldo que podemos recuperar. Cuando queríamos trabajar en el proyecto, descargábamos el último y nos poníamos a completar los métodos que faltaban por implementar.

Parte del Proyecto	Horas estimadas	Horas aproximadas
Diseño	18(cada miembro)	22
Diagrama de Secuencia	6	8
Aspectos gráficos (Estética)	6	3
Codificación	80	100
Pruebas	8	16
Corrección	3	1

Con aspectos gráficos nos referimos a la forma de mostrar la información. Entra la distribución del contenido, el añadir el thread.sleep, crear la historia, pensar nombres… La estética del juego.

Diseño

En este apartado comentaremos la evolución de los diagramas, desde el que elaboramos al empezar con el proyecto hasta el último que ha resultado después de tener la aplicación completamente terminada. Nos resultaría imposible poner aquí los diagramas completos, es por eso que hemos decidido incluirlos en un anexo al final de este documento.

Ha habido bastantes cambios significativos desde nuestro primer diseño hasta el último.

Diagrama de Clases

El número de clases del diagrama inicial y del final es el mismo, sin embargo hay una clase que se ha eliminado porque no era necesaria y otra nueva que hemos tenido que añadir. Este es el caso de Contraseña. Tenía un atributo de tipo String, un getter y un método comprobarContraseña(). Una clase para sólo esto nos pareció prescindible, y es por eso que la eliminamos. Por otra parte, nos pareció interesante añadir la clase ListaCartas, para que el usuario tuviera la posibilidad de tener más de una carta, ya que hace más entretenido el juego y da más opciones.

Hay métodos que los hemos eliminado del diagrama de clases inicial, porque tampoco los necesitábamos, o que los hemos cambiado de sitio. Por ejemplo, al eliminar la clase contraseña, su método comprobarContraseña() lo movimos a Juego. En Casilla nos pasa lo mismo, el recuperarVida() lo trasladamos a la clase Carta. De Casilla también eliminamos los métodos iniciarCasilla(), setEntrenador() y obtenerIdCasilla(). Todas las casillas se inicializan en Mapa con los valores que nosotros hemos configurado.

En el diagrama inicial teníamos que Carta era la que atacaba, ahora será la casilla la que tenga los métodos luchar, y si se trata de una CasillaElite o CasillaLider, existirán los métodos lucharConElite() o lucharConLider(). Por esa razón tuvimos que quitar de la clase Carta los métodos atacar(), calcularDanoAtaque() y calcularDefensa().

Tenemos clases a las que en un principio les dimos muy poca importancia, y que finalmente, han resultado ser más completas de lo que esperábamos. Un buen ejemplo es la clase Combate. En un principio sólo tenía los atributos carta1 y carta2 y el método simular combate. Esto es porque pensábamos hacer que la simulación fuera una única fórmula matemática para todos, concretamente la que utiliza el Pokémon original, sin embargo, finalmente, la hemos implementado de una forma más compleja. Para decidir la forma en la que se ataca, cuando se ataca y qué daño se hace, hemos recurrido a fórmulas estadísticas y a una matemática más compleja. La clase nos ha salido con nuevos atributos y dos nuevos métodos, asignarFactoresDeDano() y cambiarVida().

Una clase importante que también cambió bastante fue Juego. Esta es la clase principal, la que llama a todo lo demás, es por tanto una MAE, ya que es única. Como métodos nuevos tenemos pedirString(), que es un escáner pensado para que el usuario pueda introducir su nombre y que este se le asigne al personaje. También tenemos un contadorPartidas(), este le usamos para cambiar el nivel de los personajes del Mapa en caso de que termine el juego y se quiera continuar jugando. Y por último tenemos imprimirOpciones(), que nos muestra nuestras posibles acciones.

Continuando con los métodos nuevos nos encontramos con la clase ListaNarraciones, que en un principio sólo tenía un buscarNarracionesPorId(). Ahora esta clase cuenta con los siguientes métodos, todos muy típicos de las listas y que también hemos usado en algún laboratorio: getters, Iterator, anadirNarracion(), eliminarNarracion(), resetear(), inicializarLista(), getTamano().

Otro cambio es que antes el Líder iba a tener la contraseña y tenía un método getContrasena(). Hemos pensado que es mejor que la contraseña la de la élite, ya que es más complicada de vencer. Es por eso que ahora cuenta con un método escribirContrasena().

Con la creación de ListaCartas tuvimos que hacer algunas modificaciones en las clases Mapa, Jugador y PersonajeDelMapa. En Mapa hemos añadido el atributo cartasRegalo, y los métodos entregarCarta() e iniciarCartasRegalo(). En Jugador hemos creado el atributo listaCartasJug y el método getListaCartas(). Por último en PersonajeDelMapa tenemos el atributo regala y sus getters y setters.

Entre los cambios también tenemos métodos que en los que hemos tenido que cambiarle la forma de acceso, por ejemplo, los getters de Carta eran privados, y hemos tenido que ponerlos como públicos. Esto mismo nos ha sucedido también con otras clases.

Diagrama de Secuencia

Hemos de decir que el diagrama de secuencia ha variado muy poco. Lo más significativo del primer diagrama al último han sido los cambios de lugar de los métodos y las clases que hemos creado o eliminado. Es por este motivo que no consideramos de interés hacer una comparación entre los dos diagramas, ya que sería demasiado redundante. A continuación pasaremos a exponer el funcionamiento del diagrama de secuencia final.

Nada más ejecutar la aplicación se llama al método iniciarJuego(). Primeramente explicaremos lo que hace este método.

iniciarJuego() toma a un jugador y comienza con la inicialización. Mediante scanners guardará en una variable strings. El primero será el nombre, pregunta por el nombre del usuario, salta el scanner mediante el método pedirString(), lo guarda en una variable y después se le asigna mediante un método setNombre(). Posteriormente se pide al usuario que elija una carta, las cartas son 4 posibles que ya están creadas, dependiendo del texto introducido se escoge una u otra y se setea. De forma que en la lista de cartas del personaje siempre va a estar esa que ha elegido. Por último se le pide que introduzca una contraseña si dispone de ella. Una vez introducida se comprueba, y si es válida se cambian los atributos de nivel de la carta y también la coordenada del personaje. A continuación se hace un getMapa() y con inicializarMapa() se inicializa el mapa.

inicializarMapa() consiste en agregar casillas al mapa. Por cada casilla se crea una carta, un personaje y se crea la casilla que contiene al personaje y a la carta. Por último se añade la casilla al Mapa con anadirCasilla(). Cuando se termina ya está creado el mapa.

A continuación hay que cargar las narraciones. Para ello el procedimiento es idéntico al anterior. Se hace un getListaNarraciones() e inicializarListaNarraciones(). Por cada narración se hace una nueva narración y se añade a la lista.

Se llama a escribirNarración(), se toma al jugador para obtener las coordenadas, y estas se convierten a integer. De este modo podemos usar ese número como id y buscarlo en la ListaNarraciones con el método buscarNarracionPorId(), y posteriormente imprimirNarracion().

Una vez de que está hecho todo esto, el jugador podrá avanzar por el mapa de casilla en casilla, a través de los cinco niveles existentes.

La clase Juego llama al método jugar(), este es un bucle infinito, porque tenemos un programa recursivo que no termina nunca, sino que sigue aumentando de dificultad a medida que vamos jugando. Este método llama a un getJugador(). Del jugador obtenemos las coordenadas que nos permiten situarlo en el mapa, y de ahí nos permite saber e n qué casilla está y que contiene dicha casilla. Cogemos la casilla y utilizamos imprimirOpciones() seguido de un scanner pedirString() para que el usuario sepa lo que puede hacer y que nos diga lo que desea hacer.

Las opciones son las siguientes:

1- Salir
2- Recuperar HP
3- Luchar Con Entrenador
4- Retroceder

Si está en un Casilla o Casilla Líder
5- Avanzar
6- Luchar Con Líder

Si está en una Casilla Élite
5- Luchar con la Élite

Lo que ocurre si pulsa 1 (salir) es que el programa finaliza. Si elige 2 (recuperar HP), la carta recupera toda su vida, eso lo hace poniendo la vidaActual como vidaMaxima. Los comandos de retroceder y avanzar lo que hacen es cambiar la coordenada, aumentando o disminuyendo la posición de la columna.

La lucha varía si se trata de un entrenador, un líder o una élite. Lo primero que se hace es un getEntrenador(), y comprobar que las vecesDerrotado() sean inferior a 3. Si se cumple se hace un getSuCarta() y ejecutamos el método combatirCon(). Este método coge la carta de nuestro oponente y la enfrenta con la nuestra. El combate se desarrolla en la clase Combate.

En el caso de estar peleando contra un líder, si se cumple lo anterior y le derrotamos hacemos un setHaGanadoAlLider() y lo ponemos a true, de forma que se nos permita avanzar. Además, el líder nos puede dar una cartaRegalo si la tiene.

Por último tenemos a la élite. Son tres combates seguidos con la misma carta. Para ganar a la élite tenemos que derrotar a los tres miembros de la élite consecutivamente sin recuperar vida. Cuando esto sucede no se hace un avanzar, sino, que como es la última casilla del nivel se pasa automáticamente a la primera casilla del primer nivel. Se pone la columna a 0 y la fila a posFila+1.

A grandes rasgos este es el funcionamiento de nuestra aplicación. Los diagramas correspondientes los pondremos adjuntos en el anexo.

Casos de prueba: Diseño de las JUnits

Antes de ponernos a implementar el código del programa, ya sabíamos dónde íbamos a tener más dificultades y dónde tendríamos que hacer más comprobaciones. Sin embargo, la implementación de las JUnits no la comenzamos hasta que tuvimos ya parte del código implementado y empezamos a ver algunos fallos.

La detección de problemas, para su futura corrección, la hicimos de diferentes modos. Hemos estado usando líneas de "imprimir", de forma que nos sacara por pantalla los datos que nos eran relevantes para comprobar que se actualizaban al pasar de un punto a otro. Cuando el problema era algo más complejo y no teníamos claro de dónde podía venir, echamos mano del "debugger". Y por último, para comprobar y certificar que la solución que hemos diseñado está libre de errores, usamos las JUnits.

A continuación expondremos los casos más conflictivos y relevantes de las pruebas unitarias ordenados por clases.

Clase Juego

Esta es la clase principal, contiene el main, y va a ser la encargada de que nuestro juego funcione.

- iniciarJuego(): Tenemos que asegurarnos de que la aplicación se ejecuta correctamente, que hace las llamadas en el orden deseado, de forma que nosotros podamos ir asignando al jugador los valores deseados a través de la consola.
- Otros métodos importantes son pedirContrasena() y comprobarContrasena(). Tenemos que saber si esa clave es válida para poder hacer un set personalizado del mapa y del nivel de la carta. Como nuestro juego es recursivo, también necesitamos el contadorPartidas(), para aumentar la dificultad una vez que hayamos ganado la primera partida.

Clase Jugador

Esta es la clase que contiene a nuestro personaje, el protagonista de la aventura. El Jugador tiene un nombre, unas coordenadas y una carta.

- Hay que estar seguros de que el jugador siempre tendrá una carta en su lista. Es por eso que se setea al iniciar el juego.

- Dos métodos muy importantes son getCoordenada() y convertirCoordenada(). La coordenada nos permite saber en qué lugar estamos y cuál es el siguiente. convertirCoordenada() va a transformar un tipo Coordenada en un int. Es muy importante que lo haga correctamente, ya que este integer actúa en realidad como un id para las casillas y las narraciones.
- El jugador tiene también los métodos avanzar() y retroceder(), aunque estos los podemos encontrar en más clases. Estos pueden dar fallos importantes en dos casos, que se esté en la primera casilla y se quiera retroceder, o que se esté en la última y se quiera avanzar. En cualquier caso, no se permitirá realizar ninguna de las dos acciones. Otros casos importantes son el querer avanzar en una casilla de tipo líder sin derrotar al líder. También debemos cerciorarnos de que el avance y el retroceso se haga acorde a las coordenadas, que las nuevas coordenadas al realizar la acción sean las correctas.
- combatirCon(): Este método también es importante. En este caso sólo va a hacer la llamada a la clase Combate, que es la que se encargará de llevarla a cabo. Este método nos dirá si hemos ganado o perdido, y consecuentemente nos subirá o bajará el nivel de la carta, dependiendo del resultado. Más adelante explicaremos la importancia del desarrollo del combate.

Clase Mapa

La clase Mapa tiene dos funciones. Por un lado va a efectuar todos los movimientos, y por otro lado, es también una "Lista de Casillas", todas las posiciones, los personajes, las cartas, están guardados en esta clase.

- aniadirCasilla(): Este es si no el más, uno de los más importantes. Nos permite crear una casilla en el mapa, que puede ser de tipo Casilla, CasillaLider o CasillaElite. En la Casilla se encuentran los personajes y cartas con sus características. Es de suma importancia comprobar que se crean de forma correcta y que se pueden obtener sus datos.
- iniciarMapa(): Este método usa el anterior. Aquí creamos los personajes y las cartas, creamos la casilla que los contiene y esa casilla la añadimos al mapa.
- entregarCarta(): Una de las características que añadimos a última hora es la opción de tener una ListaCartas, así que también tenemos que comprobar que los personajes que dan cartas realmente las den, y que una vez que haya sido entregada, el protagonista la tenga en su lista.
- existenCoordenadas(): No queremos salirnos del mapa ni que apuntemos a una casilla no existente (null). Es por eso que comprobamos la existencia de las coordenadas, para evitar fallos. Nos aseguramos de que este método no falle.

Clase Carta

Durante el juego vamos a tener una o varias cartas y nos vamos a enfrentar contra otras cartas. La clase carta tiene atributos de ataque, defensa, velocidad, vida… Tenemos 4 tipos de Carta, sus atributos son varían dependiendo del tipo. Hay tres cosas que tenemos que comprobar para que no haya errores:

- La vida de una carta no puede ser negativa. Cuando llega a 0 o a un número inferior el combate termina.

- El nivel de una carta tampoco puede ser negativo. Si esto ocurriera, en vez de provocar daño a su oponente, le estaría dando vida, ya que la resta de un número negativo es una suma. Para solucionar esto, hemos decidido establecer un nivel mínimo, ya que si llega a nivel 0 o uno muy bajo, no podría nunca ganar el juego, al pelear contra el personaje más inferior lo más probable es que siguiera siendo abatido.
- Comprobar que al subir o bajar de nivel cambian tanto el nivel como los atributos. Esto se comprueba o mediante la fórmula si queremos saber exactamente el número que vamos a obtener o comprobando que los atributos son mayores al subir de nivel y menores al descender.

Clase ListaCartas

El jugador va a poder elegir entre una o más cartas. Esto es una ventaja, ya que dependiendo del adversario puede elegir una estrategia u otra. Las pruebas unitarias son las propias de una lista cualquiera.

- Siempre tiene que haber al menos una carta en la lista. Nunca puede ser una lista vacía. La carta se inicializa al principio de la partida, ya que es obligatorio elegir, y como no hay ningún método de eliminar carta disponible, no debería ser posible que el usuario se quede sin cartas. El caso que se puede dar es que las cartas no tengan vida, y lo que pasará es que si no la recupera, perderá el combate. Una de las pruebas podría ser que funcione el método recuperarVida(), pero ese pertenece a la clase anterior.
- anadirCarta(): Este es un método común de lista, hay que comprobar que al añadir una carta el tamaño de la lista aumenta.
- buscarCartaPorId(): Otro método que se puede encontrar en cualquier lista. Comprobar que la búsqueda se realiza correctamente, tanto si existe ese elemento como si no está en la lista.
- pedirInt(): Cuando tenemos más de una carta en la lista, podemos elegir con cuál deseamos luchar. Nos tenemos que asegurar que la elección se realiza de forma correcta.

Otras pruebas de interés

A continuación expondremos algunas pruebas que también consideramos significativas, pero de una importancia menor a las mencionadas con anterioridad.

- Las narraciones funcionan con un id, tenemos que asegurarnos que al pasar el convertir una coordenada a integer y usarlo como id nos aparezca por pantalla la narración que esperábamos.
- La clase combate nos dio algún que otro problema. Hay que comprobar que las cartas se hagan daño al atacar, que no ataquen con daño cero. También es importante que la carta ataque al oponente, que en el primer diseño, nuestra carta se atacaba a sí misma. Y como hemos comentado antes, que siempre se resten, que no ocurra que una carta le da vida a la otra (como ocurría cuando esta tenía nivel y atributos negativos).
- Que si estamos en una casilla de tipo líder o de tipo élite, que se cumplan las reglas determinadas para estas casillas. Es decir, que no se puedan esquivar mediante el comando avanzar.
- Combates infinitos. Al principio no pusimos límite de combate, posteriormente establecimos un contador de combates ganados para cada personaje, de forma que si derrotas 4 veces a un

entrenador, este no querrá volver a pelear. Esto es para que el usuario no aumente de niveles contra un entrenador débil y después pueda pasarse el juego como si nada.

Como ya hemos dicho al principio, estas son las pruebas más relevantes para el correcto funcionamiento de nuestro proyecto. Todas las demás pruebas que faltan están implementadas en el diseño final.

Excepciones

En este proyecto hemos usado las siguientes excepciones:

- **ArithmeticException:** Esta excepción la hemos usado en la clase Combate. Esta excepción se lanza cuando ocurre una condición aritmética excepcional.
- **ArrayIndexOutOfBoundsException:** La usamos en la clase Mapa. El objetivo de esta excepción es que salte cuando se trate de acceder al array con un índice no válido. Un ejemplo sería si queremos acceder al array con un índice negativo o mayor a su longitud. Esta excepción la podemos introducir en cualquier lista.
- **NumberFormatException:** Esta la usamos en la clase Juego. El motivo es que aquí transformamos un String en un integer, pero ¿y si ese String no es un número? Esta excepción es lanzada para indicar que la aplicación ha intentado convertir un String a un entero, pero que la cadena de caracteres no tiene el formato apropiado.
- **InterruptedException:** Esta excepción la utilizamos cuando tenemos un thread.sleep(). Se lanza cuando un hilo está esperando, en modo sleep u ocupado, y además el hilo ha sido interrumpido antes o durante su actividad. Si el hilo ha sido interrumpido se lanza la excepción.

Aspectos más destacables de la implementación

El juego en sí es bastante modular, y nos permitiría incluso hacer uno diferente cambiando muy pocas cosas. Ese es un punto que podría jugar a favor, ya que es reutilizable.

Una clase realmente interesante es Combate. En un principio habíamos pensado en algo mucho más sencillo, pero finalmente se optó por utilizar métodos estadísticos para decidir quién ataca primero, incluso se pueden atacar varias veces seguidas. De todos modos, el que más daño provoca tiene una menor velocidad, por lo que golpea menos, mientras que el que tiene menos daño es más rápido, y le permite atacar antes y más veces consecutivas. Es una forma de que esté nivelado. También tenemos que decir que dependiendo de los tipos que se enfrentan, las posibilidades de ganar varían. No es lo mismo que un Fuego se enfrente contra un Hoja a que se enfrente contra un Agua.
La estadística hace que los combates sean un tanto impredecibles, ya que no puedes estar seguro al 100% de quién va a ganar. Las pruebas son por lo tanto difíciles de hacer.

La contraseña es también una peculiaridad de nuestro proyecto. No sabíamos cómo guardar la partida, así que decidimos que nos guardara el nivel de la carta y la posición del juego en la cual nos habíamos

quedado. De este modo se puede volver a jugar desde un punto concreto una vez de que se haya cerrado la aplicación.

El acceso a las narraciones lo hemos hecho obteniendo la posición del jugador y convirtiendo esa coordenada a String. Otra forma podría haber sido introduciéndola dentro de la propia casilla. De esta forma podemos modificar las narraciones sin tener que modificar la casilla.

La lista de cartas fue una idea de última hora. Nos obligó a cambiar algunos accesos, ya que anteriormente el jugador tenía una carta en vez de una lista. Creemos que esto permite dar un mayor dinamismo al juego, en vez de jugar siempre con el mismo personaje.

En Mapa podemos destacar cómo las cartas de los personajes toman su nivel en base a las veces que lleves jugando. Es decir, si es la primera vez que se juega tiene un nivel determinado, pero si es la segunda, tiene un multiplicador para que el nivel se complique un poco más. No sería lógico volver a empezar el juego con una carta en nivel 200 y luchar contra cartas del nivel 5.

Para no tener que estar haciendo continuamente instanceOf para saber de qué tipo es la carta y cómo será el combate, tenemos el método getTipo, que nos ahorra mucho texto.

Por lo demás no tenemos mucho más que destacar. El Mapa es una matriz bidimensional, las listas son casi iguales a las que hemos hecho en los laboratorios, y la herencia la hemos aplicado en Carta y en Casilla.

Conclusiones

Para realizar este proyecto hemos dedicado mucho tiempo. Desde un principio hemos sabido lo que íbamos a diseñar y nos hemos puesto unos objetivos reales y posibles, hemos tenido los pies en el suelo en lugar de divagar con nuevas ideas. El cumplir con los objetivos intermedios en el plazo estipulado nos ha ayudado a construir un proyecto sólido, y visto el resultado, parece ser que nos hemos organizado bien.

Valoración Crítica

Desde nuestro grupo, todos los miembros estamos satisfechos con el trabajo realizado y el resultado final. Hemos invertido horas en su desarrollo, pero ha compensado. Tenemos un juego en el que hemos incluido todos los aspectos estudiados en clase, y además hemos aprendido algún que otro comando de Java que hasta la fecha desconocíamos.

Hemos diseñado los diagramas a tiempo, y hemos tratado de que tuvieran el mayor contenido y, la mayor información sobre nuestro proyecto, para que se nos pudiera guiar de la mejor forma posible. Sabemos que tienen una gran importancia, ya que trabajamos a partir de ellos. Desde el primer diagrama hasta el último hay bastante diferencia, sin embargo, lo más importante para el funcionamiento del juego se ha mantenido casi intacto.

Los objetivos se han cumplido en su totalidad, de hecho ya lo hemos comentado en la página 5 de esta memoria. Desde un principio nos preocupamos por el tiempo, por organizarlo lo mejor posible y por ponernos tareas y objetivos a corto plazo.

Puntos fuertes y principales problemas

Los puntos fuertes también los hemos comentado, creemos que es un juego modular, al que se le puede cambiar la historia y los personajes fácilmente para narrar otra diferente. También pensamos que hay elementos que son reutilizables en otros proyectos, por ejemplo clases como Mapa, Casilla, ListaNarraciones, Narracion o Carta se pueden usar para otros proyectos totalmente diferentes. También tenemos confianza en la estabilidad del juego. El código que hemos utilizado y los nombres de los métodos y de las variables no son complejos, se podría retomar en un futuro y saber su funcionamiento. Otro punto interesante es que está abierto a nuevas posibilidades, así como a última hora decidimos añadir una Lista de cartas, se podrían añadir objetos que mejoren algún atributo de la carta o cualquier cosa que se nos ocurra.

En este proyecto no todo nos ha venido rodado, también hemos tenido nuestros problemas, que hemos solucionado mediante la búsqueda en Internet, consultando a otros miembros de nuestro grupo o preguntando al profesorado. A continuación citaremos algunos de los problemas que más quebraderos nos han dado:

- **Combate:** El combate nos dio unos cuantos problemas. Las primeras pruebas eran que una de las cartas no atacaba, el daño que causaba era cero. Lo ajustamos, cambiamos algunos valores de la clase y lo volvimos a probar. Ahora el problema que aparecía era que nuestra carta causaba daño, pero se atacaba a sí misma. Volvimos a revisar el código y corregir el problema.
  Al seguir jugando, si enfrentábamos a una carta con su opuesta, ejemplo Agua con Fuego, resulta que siempre ganaba el que más ventaja tuviera, en este caso el Agua. Problema, que no se nos permite continuar la aventura porque siempre perdía. Esto lo solucionamos cambiando los factores de daño, ajustándolos hasta llegar un punto en el que en tal caso tuviera más ventaja, pero no tanta, de este modo hacía más daño, pero la diferencia no era tan abismal y daba opción a la victoria. No podíamos tener un juego en el que no se pudiera ganar.
- **Mapa:** Aquí el problema principal estaba en las coordenadas. El avanzar y retroceder fallaban, las narraciones por consiguiente tampoco coincidían con la casilla y con los personajes. Para identificar el problema, lo hicimos de un método bastante rudimentario, mediante el comando de imprimir por pantalla. En este caso imprimíamos la posición de la fila y de la columna. El problema fue en que en unas partes del código llamábamos columna a la fila y en otra parte estaba al revés. Mediante la reescritura de las coordenadas y dejar claro qué venía antes, si la fila o la columna, solucionamos el problema y conseguimos que se avanzara correctamente por el array y que se mostraran las narraciones correctas en las casillas adecuadas.
- **Scanner:** La función scanner nos dio unos cuantos problemas al principio, y es que no nos reconocía lo que escribíamos. Mejor dicho, el scanner guardaba en la variable el valor que se le pasaba por

teclado, pero después no funcionaba correctamente y provocaba un fallo en la aplicación. Al parecer eso fue porque usábamos unas cuantas veces esa función pero en puntos distintos, y el problema ocurría cuando se cerraba. Dejando los scanner abiertos no hay ningún problema y el juego funciona correctamente.

Conclusiones extraídas

Las conclusiones más importantes que hemos obtenido de la elaboración de este proyecto son las siguientes:

- Los diagramas son fundamentales para hacer un buen proyecto. Son nuestra guía y nos van a decir desde un principio qué métodos vamos a necesitar. Aunque después vayan cambiando y nos demos cuenta de que hay métodos que estarían mejor en otra clase, que hay otros que no hemos puesto pero que son necesarios, que observemos que otros son prescindibles, o que los accesos no son los correctos y hay que cambiar los permisos, hay que tener una base. Nuestro diagrama ha cambiado, pero los métodos más importantes siguen intactos.
- Es mejor poner unos objetivos sencillos a corto plazo y fáciles de cumplir antes que pensar en un diseño complejo y lejano. En nuestro caso hemos conseguido cumplir todos los objetivos que nos exigimos al principio.
- El trabajar en un grupo en el que los compañeros cumplan con su trabajo y se preocupen por el proyecto es importante. En nuestro grupo hemos estado muchas horas trabajando, tanto en equipo como de forma individual, y se han estado aportando múltiples ideas y soluciones. También nos hemos ayudado entre nosotros.
- La apariencia la hemos dejado para el final. El mostrar el texto por consola de una forma ordenada es lo último que hemos hecho. Primero hay que asegurarse que la aplicación funciona como es debido.
- Este tipo de actividades nos ayudan a aplicar los conceptos aprendidos en clase, pero además nos exige buscar información por nuestra cuenta cuando queremos hacer algo que no nos han explicado. Con este proyecto hemos aprendido también la dificultad que conlleva diseñar una aplicación desde cero.

Planificación vs. Resultado

Estamos satisfechos con ambas partes. El resultado ha sido el que esperábamos al principio. De tiempo hemos andado bastante bien y hemos cumplido con lo que pretendíamos.

Si hay algo que podamos decir es que de lo que se planea a lo que sale en realidad hay diferencia. De hecho, hemos acabado haciendo más horas de las que en un principio esperábamos. Por el camino nos han aparecido contratiempos, métodos que no funcionaban como deberían de hacerlo. Sin embargo los hemos conseguido solucionar a tiempo. Creemos que nos hemos organizado bien y estamos contentos.

¿Qué más podríamos haber incluido?

Hemos hecho una lista de las posibilidades que tiene nuestro proyecto. Seguimos pensando que para el tiempo del que disponemos y que para lo que se nos pide en esta asignatura, la solución que hemos diseñado es suficiente. Sin embargo vamos a hacer una lista de algunos añadidos que podría tener nuestro proyecto si dispusiéramos de más tiempo y las exigencias fueran distintas.

- Nuestro juego ha sido lineal. Si cogemos el juego Pokémon vemos que el jugador se puede desplazar en dos dimensiones a cualquier casilla que sea adyacente a la actual. Podríamos haber diseñado ese tipo de movimiento, o incluso una matriz de tres dimensiones por si hay edificios de más de una planta.
- Ataques variados. A última hora metimos la opción lista de cartas, tampoco hubiera sido descabellado el añadir más de un ataque a la carta. Claro, que nos harían falta una lista de ataques, una clase ataque, atributos nuevos, etc. Hubiera sido demasiado complejo para el tiempo del que disponíamos. Pero sería una característica interesante, ya que le daría más vida al juego y no sería tan monótono y repetitivo. En tal caso se podría también hacer para que los ataques no fueran automáticos, si no que a cada golpe hubiera que elegir un ataque, dando la posibilidad de repetir ataques o cambiarlos.
- Otra opción que hemos comentado antes en este documento son los objetos. No hubiera sido difícil hacer que, al igual que los líderes pueden dar cartas, que los entrenadores pudieran dar alguna recompensa. Tal vez alguna poción que restaurase la vida o que aumentara alguna habilidad, como por ejemplo la velocidad o el ataque. O, en el caso de que haya ataques variados, algún objeto que permita añadir un ataque a la lista de ataques de una carta.
- Y para terminar, la estética. No se nos pedía, porque esta asignatura se trata del diseño e implementación, pero sí que hubiera sido más bonito el hacer un ejecutable con una ventana que no fuera puramente consola. Tal vez incluso con algún sonido de fondo.

2013

PMOO ANEXO

Mikel Barcina
Mikel Dalmau
Jose Ángel Gumiel
Begoña Mtnez. De Marañón
Jonatan Pérez

Anexo

En las siguientes páginas se incluirá el material auxiliar del proyecto. Insertaremos los diagramas de clases y de secuencia utilizados y citaremos algunas de las fuentes en las que encontramos información.

En el archivo con extensión zip está también el juego, tanto en su versión editable como en versión compilada, con un ejecutable. Además de más material utilizado.

Actas de reunión

No podemos facilitar las actas de reunión porque no las hemos utilizado. En nuestro grupo hay días en los que hemos estado reunidos todos y otros en los que han estado los que han querido. No obstante, el no estar presentes en un mismo lugar no indica que no se comprometan con el trabajo. Todos los miembros hemos estado informados del trabajo que estaba ya realizado, el que faltaba por hacer y los puntos críticos en los que habían surgido problemas. Sabíamos lo que había que hacer.

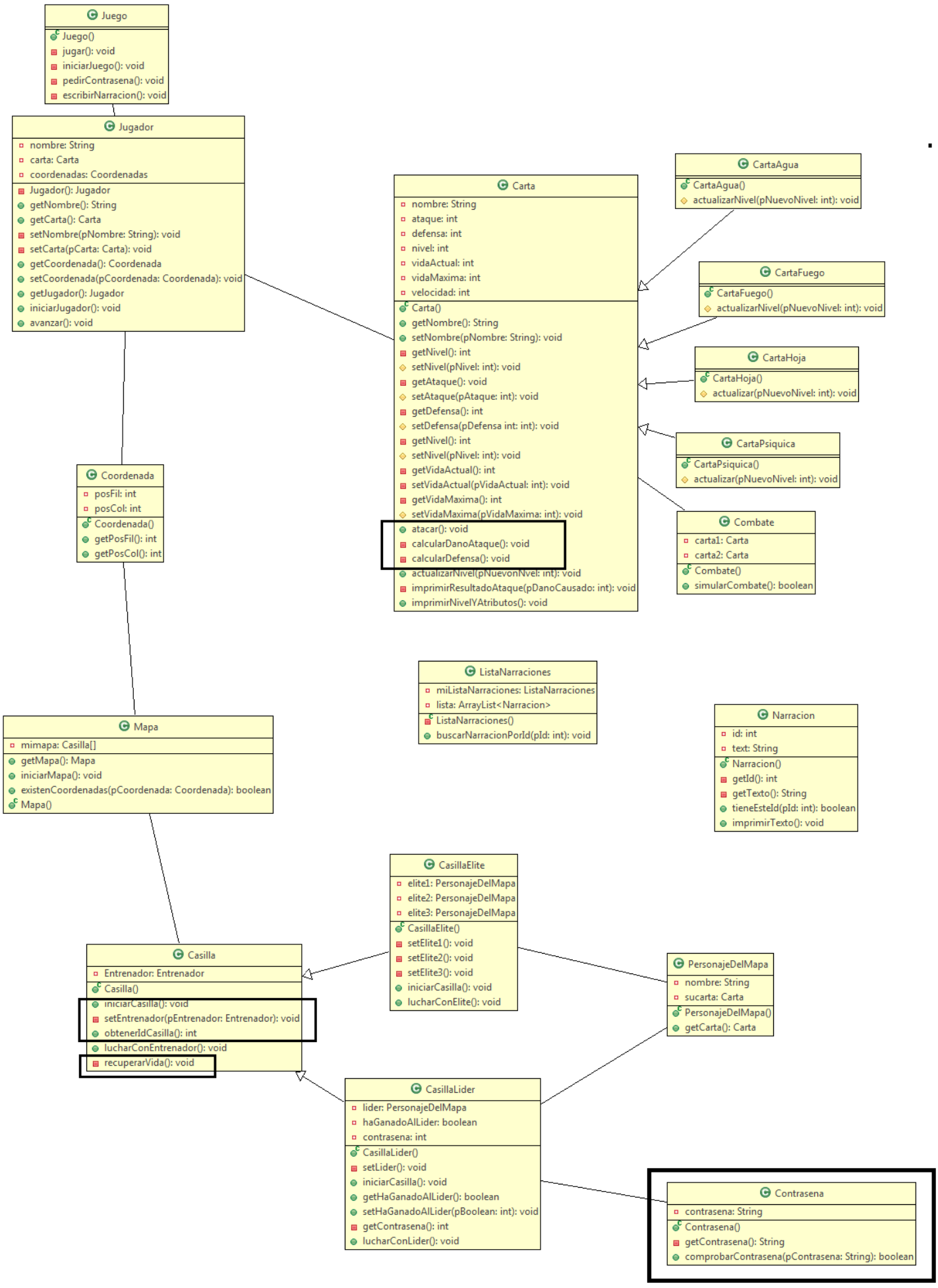
Nuestra prioridad no fue el reunirnos, sino el trabajar en equipo y que todos nos comprometiéramos con las tareas asignadas. Cada uno podía elegir en qué momento se ponía a trabajar, pero tenía que cumplir con los objetivos y las fechas acordadas.

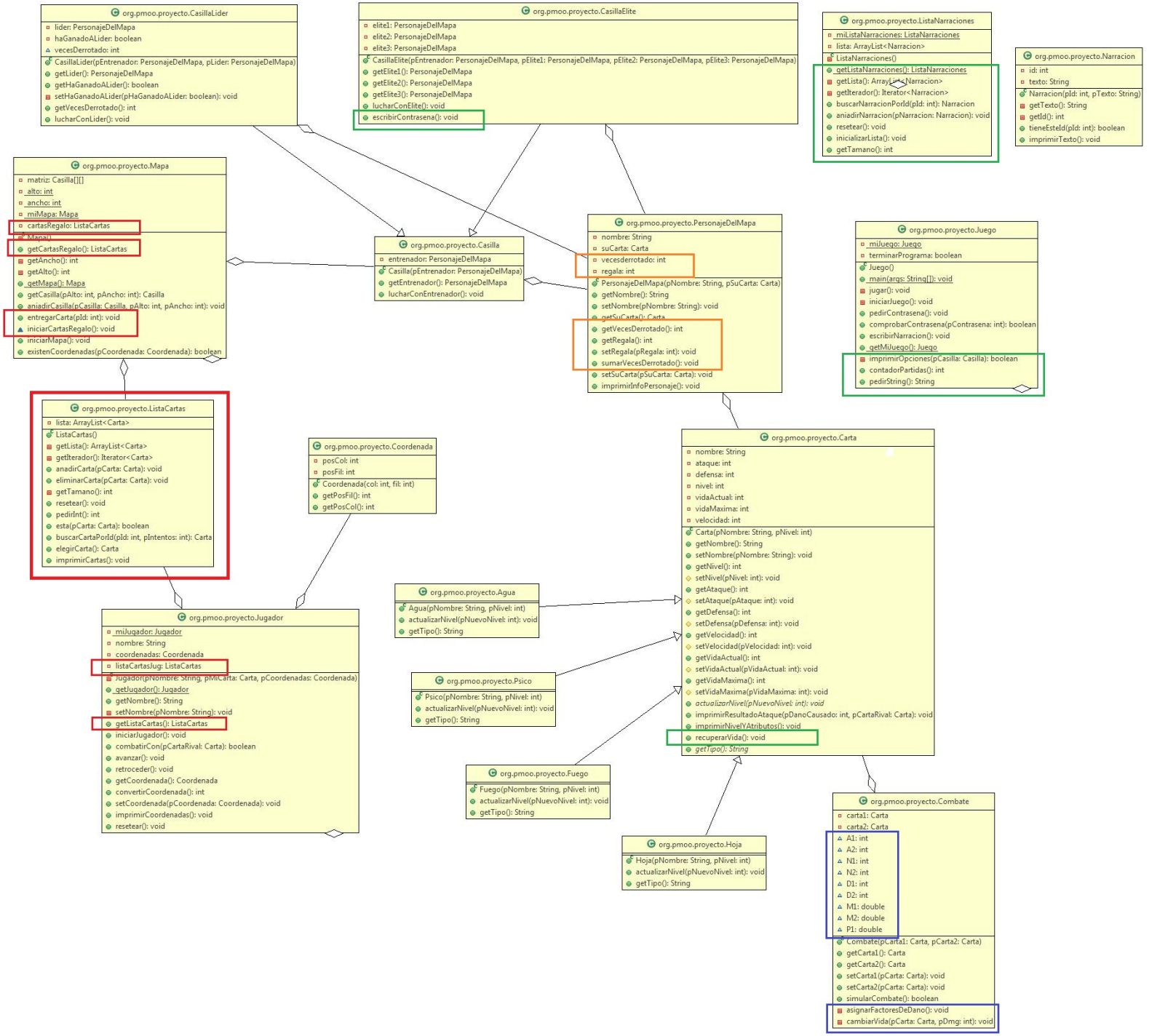
Mientras haya formalidad y se cumpla lo acordado, no hay necesidad de asistir en persona a las reuniones ni de establecer actas. La comunicación la hemos llevado a cabo en clase, en los laboratorios, en el aula de ordenadores (cuando hemos quedado) y a través de mensajería instantánea. No hemos tenido ningún problema. El trabajo se ha ido actualizando a través de una carpeta compartida en DropBox, como ya hemos comentado anteriormente.

Bibliografía

No hemos utilizado libros, pero sí que hemos usado Internet para encontrar solución a nuestros problemas, las páginas que más nos han ayudado son:

- http://docs.oracle.com Portal oficial de Oracle en el que podemos encontrar información sobre los comandos de Java que necesitemos.
- http://stackoverflow.com/ Una especie de foro. En él los usuarios preguntan y obtienen respuesta. No hemos tenido necesidad de abrir ningún hilo, ya que hemos encontrado lo que necesitábamos.


Juego
Juego()
jugar(): void
iniciarJuego(): void
pedirContrasena(): void
escribirNarracion(): void
Jugador
nombre: String
carta: Carta
coordenadas: Coordenadas
Jugador(): Jugador
getNombre(): String
getCarta(): Carta
setNombre(pNombre: String): void
setCarta(pCarta: Carta): void
getCoordenada(): Coordenada
setCoordenada(pCoordenada: Coordenada): void
getJugador(): Jugador
iniciarJugador(): void
avanzar(): void
Carta
nombre: String
ataque: int
defensa: int
nivel: int
vidaActual: int
vidaMaxima: int
velocidad: int
Carta()
getNombre(): String
setNombre(pNombre: String): void
getNivel(): int
setNivel(pNivel: int): void
getAtaque(): void
setAtaque(pAtaque: int): void
getDefensa(): int
setDefensa(pDefensa int: int): void
getNivel(): int
setNivel(pNivel: int): void
getVidaActual(): int
setVidaActual(pVidaActual: int): void
getVidaMaxima(): int
setVidaMaxima(pVidaMaxima: int): void
atacar(): void
calcularDanoAtaque(): void
calcularDefensa(): void
actualizarNivel(pNuevonNvel: int): void
imprimirResultadoAtaque(pDanoCausado: int): void
imprimirNivelYAtributos(): void
CartaAgua
CartaAgua()
actualizarNivel(pNuevoNivel: int): void
CartaFuego
CartaFuego()
actualizarNivel(pNuevoNivel: int): void
CartaHoja
CartaHoja()
actualizar(pNuevoNivel: int): void
CartaPsiquica
CartaPsiquica()
actualizar(pNuevoNivel: int): void
Combate
carta1: Carta
carta2: Carta
Combate()
simularCombate(): boolean
Coordenada
posFil: int
posCol: int
Coordenada()
getPosFil(): int
getPosCol(): int
ListaNarraciones
miListaNarraciones: ListaNarraciones
lista: ArrayList<Narracion>
ListaNarraciones()
buscarNarracionPorId(pId: int): void
Narracion
id: int
text: String
Narracion()
getId(): int
getTexto(): String
tieneEsteId(pId: int): boolean
imprimirTexto(): void
Mapa
mimapa: Casilla[]
getMapa(): Mapa
iniciarMapa(): void
existenCoordenadas(pCoordenada: Coordenada): boolean
Mapa()
CasillaElite
elite1: PersonajeDelMapa
elite2: PersonajeDelMapa
elite3: PersonajeDelMapa
CasillaElite()
setElite1(): void
setElite2(): void
setElite3(): void
iniciarCasilla(): void
lucharConElite(): void
Casilla
Entrenador: Entrenador
Casilla()
iniciarCasilla(): void
setEntrenador(pEntrenador: Entrenador): void
obtenerIdCasilla(): int
lucharConEntrenador(): void
recuperarVida(): void
PersonajeDelMapa
nombre: String
sucarta: Carta
PersonajeDelMapa()
getCarta(): Carta
CasillaLider
lider: PersonajeDelMapa
haGanadoAlLider: boolean
contrasena: int
CasillaLider()
setLider(): void
iniciarCasilla(): void
getHaGanadoAlLider(): boolean
setHaGanadoAlLider(pBoolean: int): void
getContrasena(): int
lucharConLider(): void
Contrasena
contrasena: String
Contrasena()
getContrasena(): String
comprobarContrasena(pContrasena: String): boolean


org.pmoo.proyecto.CasillaLider
lider: PersonajeDelMapa
haGanadoALider: boolean
vecesDerrotado: int
CasillaLider(pEntrenador: PersonajeDelMapa, pLider: PersonajeDelMapa)
getLider(): PersonajeDelMapa
getHaGanadoALider(): boolean
setHaGanadoALider(pHaGanadoALider: boolean): void
getVecesDerrotado(): int
lucharConLider(): void
org.pmoo.proyecto.CasillaElite
elite1: PersonajeDelMapa
elite2: PersonajeDelMapa
elite3: PersonajeDelMapa
CasillaElite(pEntrenador: PersonajeDelMapa, pElite1: PersonajeDelMapa, pElite2: PersonajeDelMapa, pElite3: PersonajeDelMapa)
getElite1(): PersonajeDelMapa
getElite2(): PersonajeDelMapa
getElite3(): PersonajeDelMapa
lucharConElite(): void
escribirContrasena(): void
org.pmoo.proyecto.ListaNarraciones
miListaNarraciones: ListaNarraciones
lista: ArrayList<Narracion>
ListaNarraciones()
getListaNarraciones(): ListaNarraciones
getLista(): ArrayList<Narracion>
getIterador(): Iterator<Narracion>
buscarNarracionPorId(pId: int): Narracion
aniadirNarracion(pNarracion: Narracion): void
resetear(): void
inicializarLista(): void
getTamano(): int
org.pmoo.proyecto.Narracion
id: int
texto: String
Narracion(pId: int, pTexto: String)
getTexto(): String
getId(): int
tieneEsteId(pId: int): boolean
imprimirTexto(): void
org.pmoo.proyecto.Mapa
matriz: Casilla[][]
alto: int
ancho: int
miMapa: Mapa
cartasRegalo: ListaCartas
Mapa()
getCartasRegalo(): ListaCartas
getAncho(): int
getAlto(): int
getMapa(): Mapa
getCasilla(pAlto: int, pAncho: int): Casilla
aniadirCasilla(pCasilla: Casilla, pAlto: int, pAncho: int): void
entregarCarta(pId: int): void
iniciarCartasRegalo(): void
iniciarMapa(): void
existenCoordenadas(pCoordenada: Coordenada): boolean
org.pmoo.proyecto.Casilla
entrenador: PersonajeDelMapa
Casilla(pEntrenador: PersonajeDelMapa)
getEntrenador(): PersonajeDelMapa
lucharConEntrenador(): void
org.pmoo.proyecto.PersonajeDelMapa
nombre: String
suCarta: Carta
vecesderrotado: int
regala: int
PersonajeDelMapa(pNombre: String, pSuCarta: Carta)
getNombre(): String
setNombre(pNombre: String): void
getSuCarta(): Carta
getVecesDerrotado(): int
getRegala(): int
setRegala(pRegala: int): void
sumarVecesDerrotado(): void
setSuCarta(pSuCarta: Carta): void
imprimirInfoPersonaje(): void
org.pmoo.proyecto.Juego
miJuego: Juego
terminarPrograma: boolean
Juego()
main(args: String[]): void
jugar(): void
iniciarJuego(): void
pedirContrasena(): void
comprobarContrasena(pContrasena: int): boolean
escribirNarracion(): void
getMiJuego(): Juego
imprimirOpciones(pCasilla: Casilla): boolean
contadorPartidas(): int
pedirString(): String
org.pmoo.proyecto.ListaCartas
lista: ArrayList<Carta>
ListaCartas()
getLista(): ArrayList<Carta>
getIterador(): Iterator<Carta>
anadirCarta(pCarta: Carta): void
eliminarCarta(pCarta: Carta): void
getTamano(): int
resetear(): void
pedirInt(): int
esta(pCarta: Carta): boolean
buscarCartaPorId(pId: int, pIntentos: int): Carta
elegirCarta(): Carta
imprimirCartas(): void
org.pmoo.proyecto.Coordenada
posCol: int
posFil: int
Coordenada(col: int, fil: int)
getPosFil(): int
getPosCol(): int
org.pmoo.proyecto.Carta
nombre: String
ataque: int
defensa: int
nivel: int
vidaActual: int
vidaMaxima: int
velocidad: int
Carta(pNombre: String, pNivel: int)
getNombre(): String
setNombre(pNombre: String): void
getNivel(): int
setNivel(pNivel: int): void
getAtaque(): int
setAtaque(pAtaque: int): void
getDefensa(): int
setDefensa(pDefensa: int): void
getVelocidad(): int
setVelocidad(pVelocidad: int): void
getVidaActual(): int
setVidaActual(pVidaActual: int): void
getVidaMaxima(): int
setVidaMaxima(pVidaMaxima: int): void
actualizarNivel(pNuevoNivel: int): void
imprimirResultadoAtaque(pDanoCausado: int, pCartaRival: Carta): void
imprimirNivelYAtributos(): void
recuperarVida(): void
getTipo(): String
org.pmoo.proyecto.Jugador
miJugador: Jugador
nombre: String
coordenadas: Coordenada
listaCartasJug: ListaCartas
Jugador(pNombre: String, pMiCarta: Carta, pCoordenadas: Coordenada)
getJugador(): Jugador
getNombre(): String
setNombre(pNombre: String): void
getListaCartas(): ListaCartas
iniciarJugador(): void
combatirCon(pCartaRival: Carta): boolean
avanzar(): void
retroceder(): void
getCoordenada(): Coordenada
convertirCoordenada(): int
setCoordenada(pCoordenada: Coordenada): void
imprimirCoordenadas(): void
resetear(): void
org.pmoo.proyecto.Agua
Agua(pNombre: String, pNivel: int)
actualizarNivel(pNuevoNivel: int): void
getTipo(): String
org.pmoo.proyecto.Psico
Psico(pNombre: String, pNivel: int)
actualizarNivel(pNuevoNivel: int): void
getTipo(): String
org.pmoo.proyecto.Fuego
Fuego(pNombre: String, pNivel: int)
actualizarNivel(pNuevoNivel: int): void
getTipo(): String
org.pmoo.proyecto.Hoja
Hoja(pNombre: String, pNivel: int)
actualizarNivel(pNuevoNivel: int): void
getTipo(): String
org.pmoo.proyecto.Combate
carta1: Carta
carta2: Carta
A1: int
A2: int
N1: int
N2: int
D1: int
D2: int
M1: double
M2: double
P1: double
Combate(pCarta1: Carta, pCarta2: Carta)
getCarta1(): Carta
getCarta2(): Carta
setCarta1(pCarta: Carta): void
setCarta2(pCarta: Carta): void
simularCombate(): boolean
asignarFactoresDeDano(): void
cambiarVida(pCarta: Carta, pDmg: int): void

actor
Juego
Jugador
Mapa
Casilla
CasillaLider
CasillaElite
PersonajeDelMapa
Entrenador
Lider
Elite
Carta
CartaFuego
CartaAgua
CartaPlanta
CartaPsiquica
ListaNarraciones
Narracion
Contraseña
Coordenadas
Combate
Jugador
Carta
Instance2
1.Jugar
2.IniciarJuego()
3.getJugador()
4.IniciarJugador()
5.setNombre()
6.setCarta()
Segun la carta que elija el jugador se llamaria a uno de los siguientes 4 metodos:
7.CartaFuego()
8.CartaAgua
9.CartaPlanta()
10.CartaPsiquica()
11.iniciarMapa()
Por cada casilla
12.iniciarCasilla()
13.setEntrenador()
14.Entrenador()
15.asignarCarta()
Se creara una nueva Carta del mismo modo que en el setCarta() de la clase Jugador
Si la Casilla es instancia de CasillaLider
16.iniciarCasillaLider()
17.setLider()
18.Lider()
19.asignarCarta()
Si la Casilla es instancia de CasillaElite
por cada personaje elite
20.iniciarCasillaElite()
21.setElite1()
22.Elite()
23.asignarCarta()
24.iniciarListaNarraciones
25.getIterador()
26.Narracion(pId,pTexto)
27.pedirContrasena()
28.comprobarContraseña(pContrasena)
Si la pass era correcta
29.setCoordenadas()
30.existenCoordenadas()
31.getCoordenadas()
32.setCoordenadas(pFila,pColumna)
33.Coordenadas(pFila, pColumna)
34.buscarNarracionPorID(pId)
35.getIterador()
36.tieneEsteId()
37.getId()
38.imprimirTexto()
39.getTexto()
Si esta en una CasillaLider
Si decide avanzar
40.getHaganadoALider()
Si le ha ganado
41.avanzar()
42.avanzar()
43.setCoordenadas(pFila,pColumna)
44.Coordenadas(pFila,pColumna)
Si decide luchar con el lider
45.lucharConLider()
46.getLider()
47.getCarta()
48.getCarta()
49.Combate(pCartaJugador,pCartaRival))
Por cada carta y hasta que una de las 2 se quede sin HP
50.simularCombate()
51.atacar()
52.calcularDanoAtaque()
53.getAtaque()
54.getVelocidad()
55.getNivel()
56.calcularDefensa()
57.getNivel()
58.getDefensa()
Si el jugador gana el combate
59.sethaGanadoALider(true)
60.imprimirContrasena()
61.getContrasena()

Si el jugador gana el combate subirá de nivel y si lo pierde bajara

Dependiendo de la Carta se realizará una llamada de las 4 siguientes:

62.CambiarNivel(pNuevoNivel)

63.cambiarNivel(pNivel)

64.setNivel(pNivel)

65.setVidaMaxima(pVidaMax)

66.setAtaque(pAtaque)

67.setVelocidad(pVelocidad)

68.setDefensa(pDefensa)

69.cambiarNivel(pNivel)

70.cambiarNivel(pNivel)

Estos 3 metodos llamaran a los mismos setter ya que la diferencia entre ellos radica en que los atributos de la carta aumentan de forma diferente segun el tipo al que pertenezca dicha carta

71.cambiarNivel(pNivel)

Si elige luchar con entrenador

72.lucharConEntrenador()

Este método es igual que lucharConLider con la salvedad de que aqui no se llama a setHaGanadoALider

Si elige recuperar la vida de la Carta

73.getCarta()

74.recuperarVida()

75.getVidaMaxima()

76.setVidaActual(pVidaActual)

Si está en una Casilla Elite

77.getNarracionPorId(pid)

78.getIterador()

79.tieneEsteId(pId)

80.getId()

81.imprimirTexto()

82.getTexto()

En este punto se le daria al jugador la opcion de luchar con la elite, luchar con un entrenador o recuperar vida

Si elige recuperar vida se llamaria a metodo recuperarVida() de la clase Carta

Si elige luchar con entrenador se llamaria al metodo lucharConEntrenador() que viene heredado de la clase Casilla

Si elige luchar con elite

Por cada personaje elite

83.lucharConElite()

84.getElite1()

85.getCarta()

86.getCarta()

87.Combate(pCartaJugador,pCartaRival)

88.simulaCombate()

Está explicado arriba como actua

Si gana a los 3

89.cambiarNivel(pNuevoNivel)

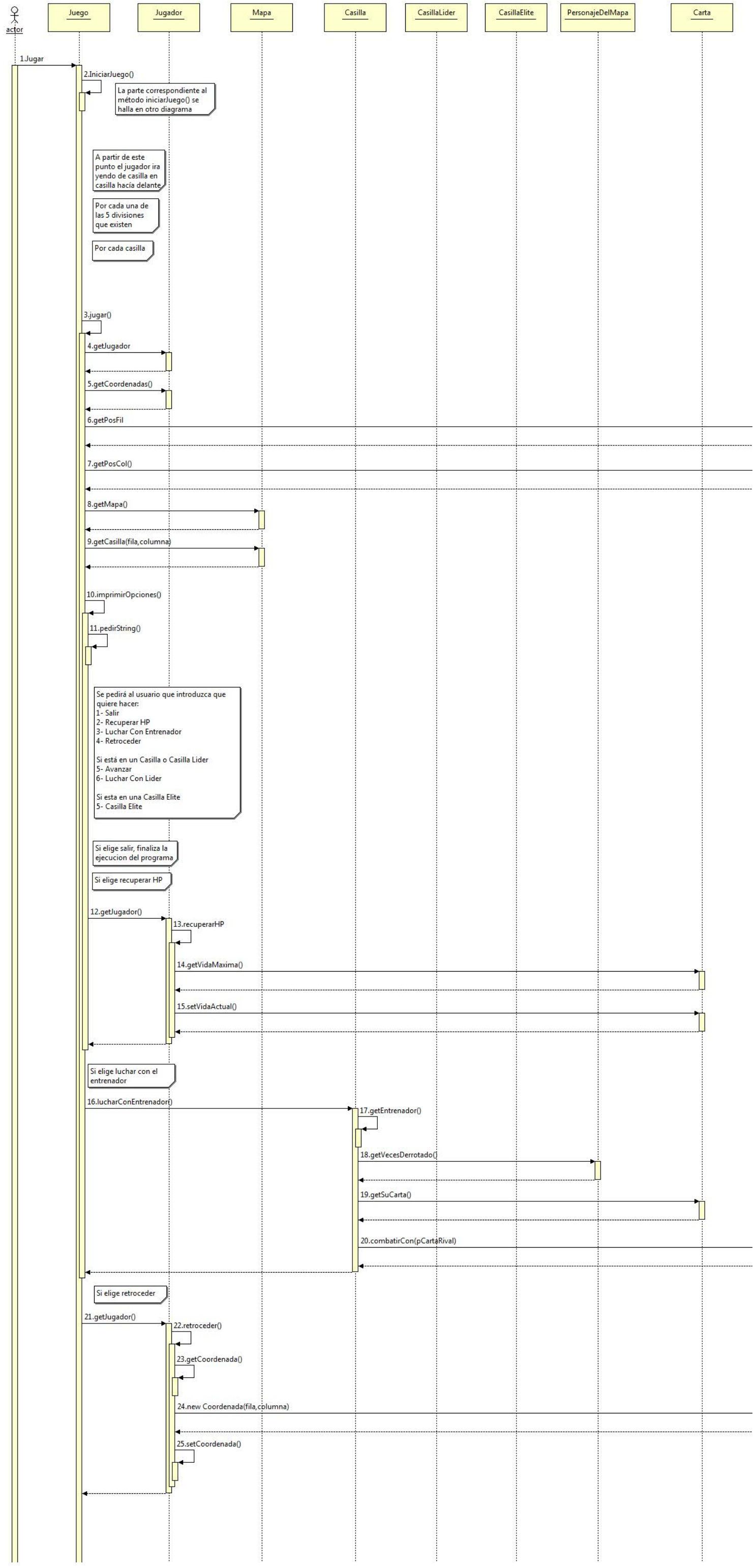
Esta explicado arriba

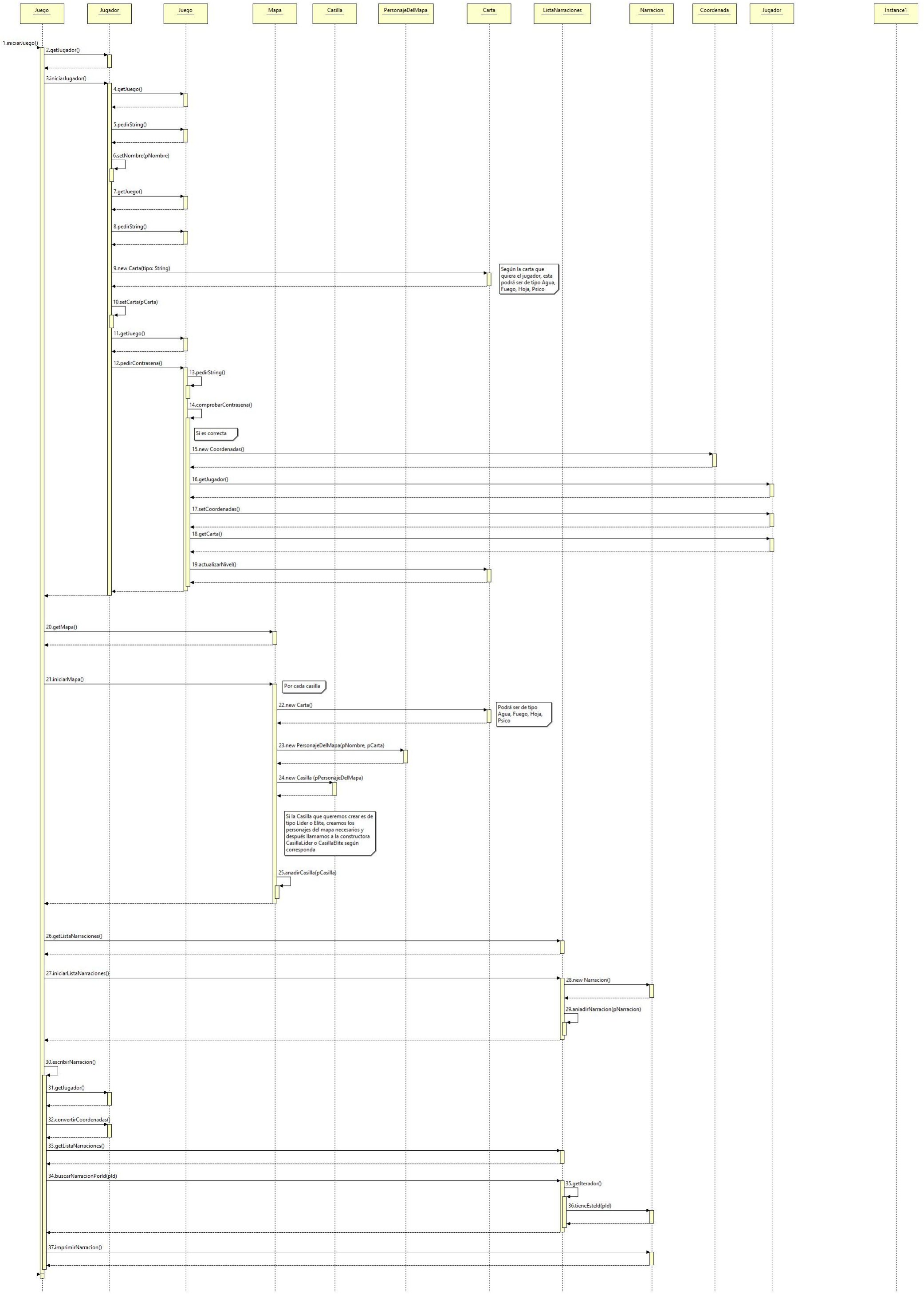
90.cambiarDeDivision()

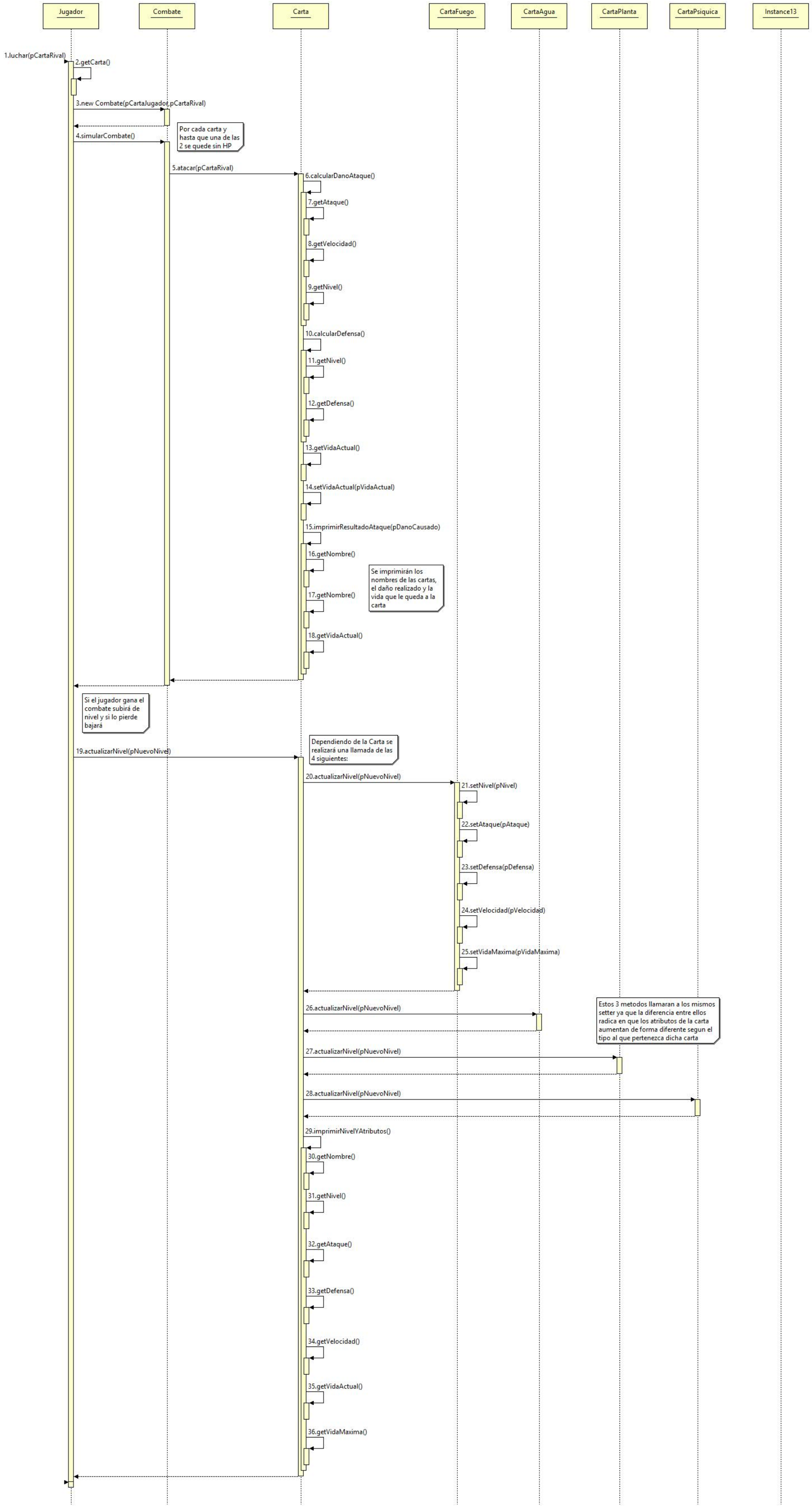
91.setCoordenadas(pFila,pColumna)

92.Coordenadas(pFila,pColumna)

Y vuelta a empezar


actor
Juego
Jugador
Mapa
Casilla
CasillaLider
CasillaElite
PersonajeDelMapa
Carta
1.Jugar
2.IniciarJuego()
La parte correspondiente al método iniciarJuego() se halla en otro diagrama
A partir de este punto el jugador ira yendo de casilla en casilla hacía delante
Por cada una de las 5 divisiones que existen
Por cada casilla
3.jugar()
4.getJugador
5.getCoordenadas()
6.getPosFil
7.getPosCol()
8.getMapa()
9.getCasilla(fila,columna)
10.imprimirOpciones()
11.pedirString()
Se pedirá al usuario que introduzca que quiere hacer:
1- Salir
2- Recuperar HP
3- Luchar Con Entrenador
4- Retroceder
Si está en un Casilla o Casilla Lider
5- Avanzar
6- Luchar Con Lider
Si esta en una Casilla Elite
5- Casilla Elite
Si elige salir, finaliza la ejecucion del programa
Si elige recuperar HP
12.getJugador()
13.recuperarHP
14.getVidaMaxima()
15.setVidaActual()
Si elige luchar con el entrenador
16.lucharConEntrenador()
17.getEntrenador()
18.getVecesDerrotado()
19.getSuCarta()
20.combatirCon(pCartaRival)
Si elige retroceder
21.getJugador()
22.retroceder()
23.getCoordenada()
24.new Coordenada(fila,columna)
25.setCoordenada()


Juego
Jugador
Juego
Mapa
Casilla
PersonajeDelMapa
Carta
ListaNarraciones
Narracion
Coordenada
Jugador
Instance1
1.iniciarJuego()
2.getJugador()
3.iniciarJugador()
4.getJuego()
5.pedirString()
6.setNombre(pNombre)
7.getJuego()
8.pedirString()
9.new Carta(tipo: String)
Según la carta que quiera el jugador, esta podrá ser de tipo Agua, Fuego, Hoja, Psico
10.setCarta(pCarta)
11.getJuego()
12.pedirContrasena()
13.pedirString()
14.comprobarContrasena()
Si es correcta
15.new Coordenadas()
16.getJugador()
17.setCoordenadas()
18.getCarta()
19.actualizarNivel()
20.getMapa()
21.iniciarMapa()
Por cada casilla
22.new Carta()
Podrá ser de tipo Agua, Fuego, Hoja, Psico
23.new PersonajeDelMapa(pNombre, pCarta)
24.new Casilla (pPersonajeDelMapa)
Si la Casilla que queremos crear es de tipo Lider o Elite, creamos los personajes del mapa necesarios y después llamamos a la constructora CasillaLider o CasillaElite según corresponda
25.anadirCasilla(pCasilla)
26.getListaNarraciones()
27.iniciarListaNarraciones()
28.new Narracion()
29.aniadirNarracion(pNarracion)
30.escribirNarracion()
31.getJugador()
32.convertirCoordenadas()
33.getListaNarraciones()
34.buscarNarracionPorId(pId)
35.getIterador()
36.tieneEsteId(pId)
37.imprimirNarracion()


Jugador
Combate
Carta
CartaFuego
CartaAgua
CartaPlanta
CartaPsiquica
Instance13
1.luchar(pCartaRival)
2.getCarta()
3.new Combate(pCartaJugador,pCartaRival)
4.simularCombate()
Por cada carta y hasta que una de las 2 se quede sin HP
5.atacar(pCartaRival)
6.calcularDanoAtaque()
7.getAtaque()
8.getVelocidad()
9.getNivel()
10.calcularDefensa()
11.getNivel()
12.getDefensa()
13.getVidaActual()
14.setVidaActual(pVidaActual)
15.imprimirResultadoAtaque(pDanoCausado)
16.getNombre()
Se imprimirán los nombres de las cartas, el daño realizado y la vida que le queda a la carta
17.getNombre()
18.getVidaActual()
Si el jugador gana el combate subirá de nivel y si lo pierde bajará
19.actualizarNivel(pNuevoNivel)
Dependiendo de la Carta se realizará una llamada de las 4 siguientes:
20.actualizarNivel(pNuevoNivel)
21.setNivel(pNivel)
22.setAtaque(pAtaque)
23.setDefensa(pDefensa)
24.setVelocidad(pVelocidad)
25.setVidaMaxima(pVidaMaxima)
26.actualizarNivel(pNuevoNivel)
Estos 3 metodos llamaran a los mismos setter ya que la diferencia entre ellos radica en que los atributos de la carta aumentan de forma diferente segun el tipo al que pertenezca dicha carta
27.actualizarNivel(pNuevoNivel)
28.actualizarNivel(pNuevoNivel)
29.imprimirNivelYAtributos()
30.getNombre()
31.getNivel()
32.getAtaque()
33.getDefensa()
34.getVelocidad()
35.getVidaActual()
36.getVidaMaxima()