FACULTAD DE INGENIERIA
Universidad de Buenos Aires

FIUBA - 75.07

Algoritmos y Programación III

Trabajo práctico 2: Al-Go-Oh!
1er cuatrimestre, 2018
Primer cuatrimestre de 2018

Alumnos:

Nombre	Padrón	Mail
MARCÓ DEL PONT, Matías	101302	matiasmdelp@gmail.com
ILLESCAS, Geronimo	102071	gero17illescas@gmail.com
DEL CARRIL, Manuel	100772	manueldelcarril@gmail.com
ROMERO VÁZQUEZ, Maximiliano	99118	maxi9614@gmail.com

Fecha de entrega final:

Tutor: Tomás Bustamante

Comentarios:

Índice

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación que implemente un juego basado en el juego de mesa presente en el manga (historieta japonesa) Yu-Gi-Oh!. Utilizando los conceptos del paradigma de la orientación a objetos y de los patrones de diseño vistos hasta ahora en el curso.

2. Supuestos

Un supuesto tomado, es que el Jugador no puede elegir en que lugar específico colocar sus cartas, es decir, dentro de la "zona de monstruos"no puede elegir si colocarlo en el medio del campo o en un extremo por ejemplo. Se ubican siempre de izquierda a derecha, y lo hace en posición ataque, para cambiarlo de posición debe esperar al siguiente turno.

También, tomamos como supuesto el hecho de que no pueda haber más de una parte de Exodia por mazo, es decir, que sólo hay una copia de cada parte de Exodia dentro del mazo. No puede suceder que haya dos piernas por ejemplo.

Otro supuesto es el funcionamiento de los sacrificios. Nuestra idea es que cada jugador debe elegir previamente a invocar un monstruo, qué monstruos de su campo decide ofrecer como sacrificio para que cuando agregue este nuevo monstruo al campo, los anteriores sean enviados al cementerio.

3. Modelo de dominio

El programa consta de las siguientes clases:

- **Campo** : es el área de juego de cada jugador, donde ubicará sus cartas. Además contiene al Mazo de dicho jugador [1].
- **Carta** : representa a una Carta del juego Al-Go-Oh; no es una clase abstracta. Es capaz de realizar su efecto o su efecto de volteo. Es instanciada para las cartas magicas, trampas o de campo del juego; para los monstruos se utiliza su clase hija, Monstruo. Puede estar boca arriba o boca abajo.
  - **Monstruo** : es el tipo de carta monstruo. Puede atacar a otros monstruos o defenderse, y al igual que su madre, realizar efectos o no.
- **Boca**: es la interfaz que implementan las dos posiciones posibles de colocar una carta (Boca arriba y boca abajo). Es la encargada de verificar que la realización de un ataque es posible o no y además es en quien Monstruo delega cuando tiene que activar efectos de volteo.
  - **BocaArriba** : implementa a Boca y representa el estado boca arriba. Permite la realización de ataques, pero no hace en los efectos de volteo.
  - **BocaAbajo** : implementa a Boca y represena el estado boca abajo. Lanza excepción cuando se quiere realizar un ataque, pero sí realiza efectos de volteo.
- **Efecto** : es la interfaz que implementan los efectos o poderes especiales que posee cualquier Carta. Tiene los métodos realizarEfecto y realizarEfectoDeVolteo, cuya diferencia es el momento en que son llamados: el realizarEfectoDeVolteo se llama durante el ataque del oponente, mientras que el otro no (Sus implementaciones dependerán de las relgas de la carta de juego).

[1] Más adelante se mostrará dicha relación en su respectivo diagrama.

  - **EfectoAgujeroOscuro** : es el efecto que posee la carta magica .ªgujero Oscuro", la cual destruye todos los monstruos del campo del jugador y el oponente. No implementa el efecto de volteo.
  - **EfectoAumentar500Ataque** : Este efecto aumenta el ataque del monstruo atacado en 500 puntos antes del calculo de daño; por ello, implementa el efecto de volteo.
  - **EfectoCilindroMagico** : corresponde al efecto de la carta mágica Çilindro Mágico", negando el ataque del monstruo atacante, e inflige el mismo daño directamente a los puntos de vida del oponente.
  - **EfectoDestruirMonstruoAtacante** : hace referencia al efecto que tiene la carta de tipo monstruo llamada Ïnsecto come-hombres". Dicha carta destruye un monstruo en el campo, y únicamente puede activarse cuando pasa de estar boca abajo a boca arriba. Puede ser activada en el momento en que el monstruo es atacado.
  - **EfectoFisura** : representa al efecto que posee la carta mágica "Fisura", que le permite al jugador destruir al monstruo boca arriba con menor ataque en el campo del oponente (en caso de haber empate, elige al azar).
  - **EfectoJinzo7** : es el efecto que contiene la carta Monstruo denominada "Jinzo 7", el cual puede atacar directamente a los puntos de vida del oponente.
  - **EfectoOllaDeLaCodicia** : éste efecto le permite al jugador tomar 2 cartas del mazo. Se encuentra en la carta Mágica .ºlla de la Codicia".
  - **EfectoSogen** : éste efecto aumenta en 500 los puntos de defensa de tus monstruos, y 200 los puntos de ataque de los monstruos de tu oponente. Se encuentra en la carta De Campo "Sogen".
  - **EfectoVacio** : hace referencia a los monstruos que no tienen efecto.
  - **EfectoWasteland** : dicho efecto aumenta en 200 puntos el ataque de tus monstruos, y 300 puntos la defensa de los monstruos de tu oponente, y corresponde a la carta De Campo "Wasteland".
  - **EfectoSacrificioDragonBlanco** : este efecto lo tienen todos los Dragones Blancos de Ojos Azules y hace referencia a su sacrificio para poder invocar al Dragon Definitivo de Ojos Azules.

- **Invocacion** : es una interfaz y representa a la forma que se debe invocar un Monstruo. Varia según su cantidad de estrellas u otro hecho en particular de cada carta. Monstruo delegará la realización de los sacrificios en alguno de los objetos que implementan esta interfaz.

  - **Invocacion1Sacrificio** : éste tipo de invocación requiere realizar el sacrificio de un monstruo del campo. Corresponde a aquellas cartas monstruo que tienen 5 ó 6 estrellas.
  - **Invocacion2Sacrificio** : dicha forma de invocar a un monstruo, requiere realizar el sacrificio de dos monstruos del campo. Corresponde a aquellos monstruos cuyo cantidad de estrellas es 7 ó más.
  - **InvocacionDragonDefinitivoDeOjosAzules** : representa a la única forma de invocar al "Dragon definitivo de ojos azules"que es sacrificando 3 dragones azules, que se encuentren en el campo de juego.
  - **InvocacionNormal** : éste tipo de invocación no requiere ningún sacrificio, y se encuentran en las cartas monstruos cuyas estrellas no superan las 4.

- **Jugador** : representa al jugador. Ataca a los Monstruos del Campo del oponente con sus Monstrous que colocó sobre su propio Campo. También coloca cartas mágicas o trampas sobre el campo y es capaz de activarlas.

- **Estado** : representa el estado del juego (ganador o no). Es la clase que decide si un Jugador gano o no.

- **Mazo** : representa una baraja de cartas del juego en cuestión, conteniendo todas las cartas que un jugador puede tomar.
- **AlGoOh** : Representa el ”organizador” o ”réferi” del juego. Es quien se encarga de llevar la cuenta de los turnos y fases de cada Jugador.
- **Fase** : Es la interfaz que implementan las distintas fases del juego. Jugador siempre antes de realizar una acción llama al método correspondiente en la fase, generando así que se lance una excepción cuando la acción es inválida. Las distintas fases implementarán estos métodos de manera distinta dependiendo de las reglas del juego. Éstas son: **FasePreparacion**, **FaseAtaque** y **FaseFinal**.
- **Posicion** : es una 'interface' y representa las posiciones que puede tener una Carta del tipo Monstruo.
  - **PosicionAtaque** : es la implementación de la posicion de ataque del Monstruo y es en quien Monstruo delega el cálculo de daño.
  - **PosicionDefensa** : es la implementación de la posición de defensa del Monstruo y su responsabilidad es la misma que la de PosicionAtaque.
- **Turno**: es una 'interface' que determina el turno en el que fue invocado un monstruo. Según éste, se pueden realizar o no, determinadas acciones, como por ejemplo, voltearlo o cambiarlo de posición. El tipo de turno puede ser: **TurnoActual** o **TurnoAnterior**.
- **HandlerCarta**: es una 'interface'. Es utilizada por la carta para poder diferenciar el comportamiento que tendrá esa carta en la interfaz gráfica según si la carta es de tipo, mágica, trampa o de campo. Dentro de las clases que implementan esta interfaz tiene un método que recibe una carta y envían el handler de la correspondiente carta. Un detalle importantes es que monstruo no implementa esta interfaz ya que al heredar de carta se diferencia de las demás por herencia y redefine el método getHandler y devuelve su propio hanlder.

4. Diagramas de clase

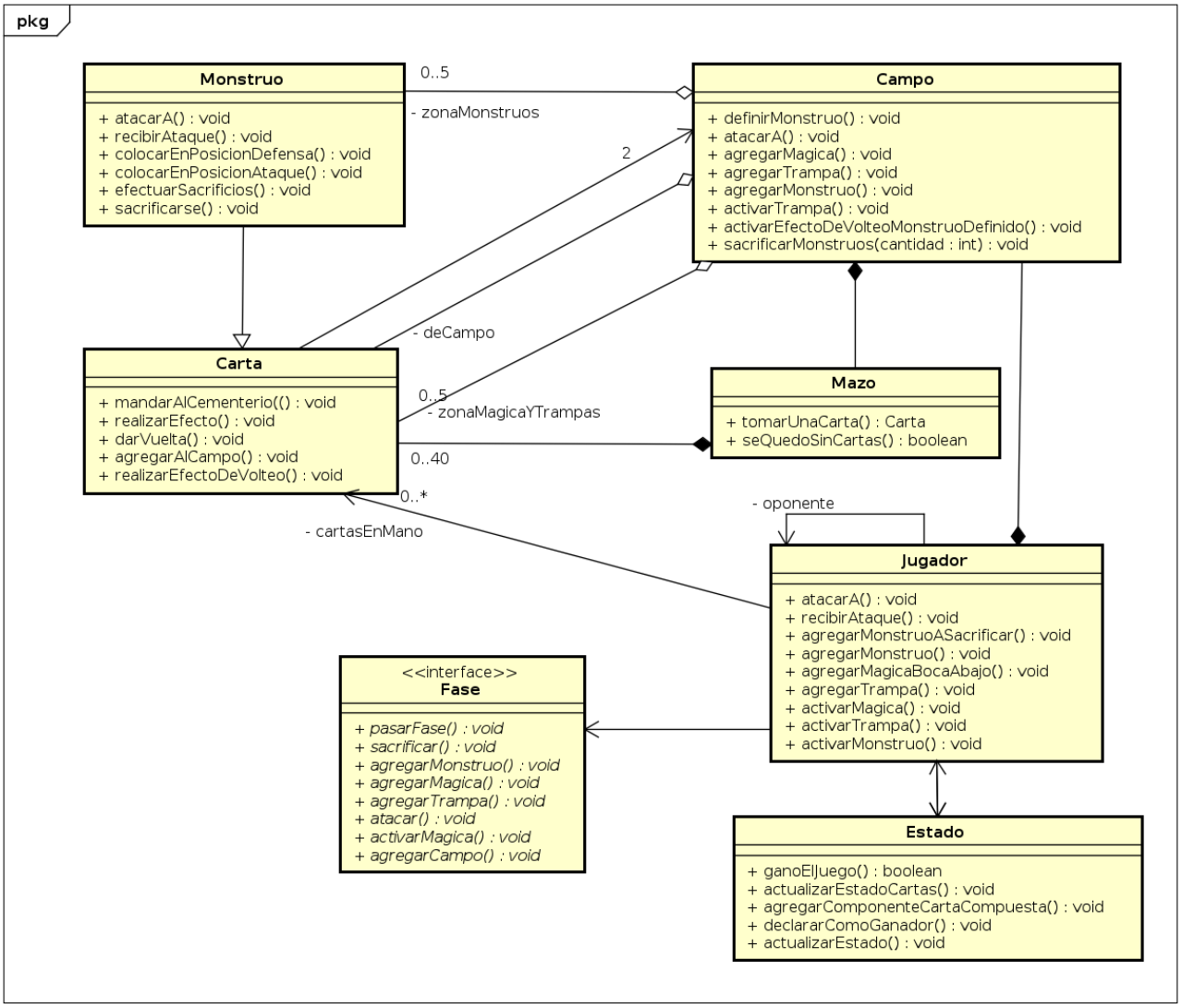

Figura 1: Diagrama de Clase principal.

En este diagrama se pueden observar las clases mas abarcativas del programa y sus relaciones.

Tenemos en primer lugar a la clase Jugador quien como dicho anteriormente coloca cartas (desde su mano) y comanda ataques. Para ello utiliza a Campo sobre quien delega la gran mayoría de estas responsabilidades. Posee también un estado, quien se encarga de saber cuando el jugador es ganador o no.

Jugador tiene como atributo una referencia a un objeto que implementa la interfaz de Fase. Cada vez que Jugador desea realizar una acción, llama al método correspondiente de Fase. El objeto de acuerdo a su implementación, levantará una excepción o no (según las reglas del juego). El Jugador re lanzará esa excepción a la clase cliente de Jugador para notificarle de la imposibilidad de usar el método en cuestión en ese momento.

Campo posee colecciones de Cartas y Monstruos para almacenar las colocaciones de Jugador.

Cada vez que Jugador realiza un ataque o activa un efecto, Campo es el encargado de localizar a la Carta en cuestión y notificarle de la acción. Monstruo finalmente realiza las acciones correspondientes.

A la hora de realizar efectos, la secuencia de mensajes es similar (Jugador le avisa a Campo de activar el efecto de un Monstruo y Campo le avisa a Monstruo. Monstruo, una vez hechas las validaciones internas, vuelve a delegar en Campo.

En las invocaciones sucede lo mismo.

Campo tiene un Mazo del cual puede sacar cartas para cuando el jugador al principio de cada turno, le pida una.

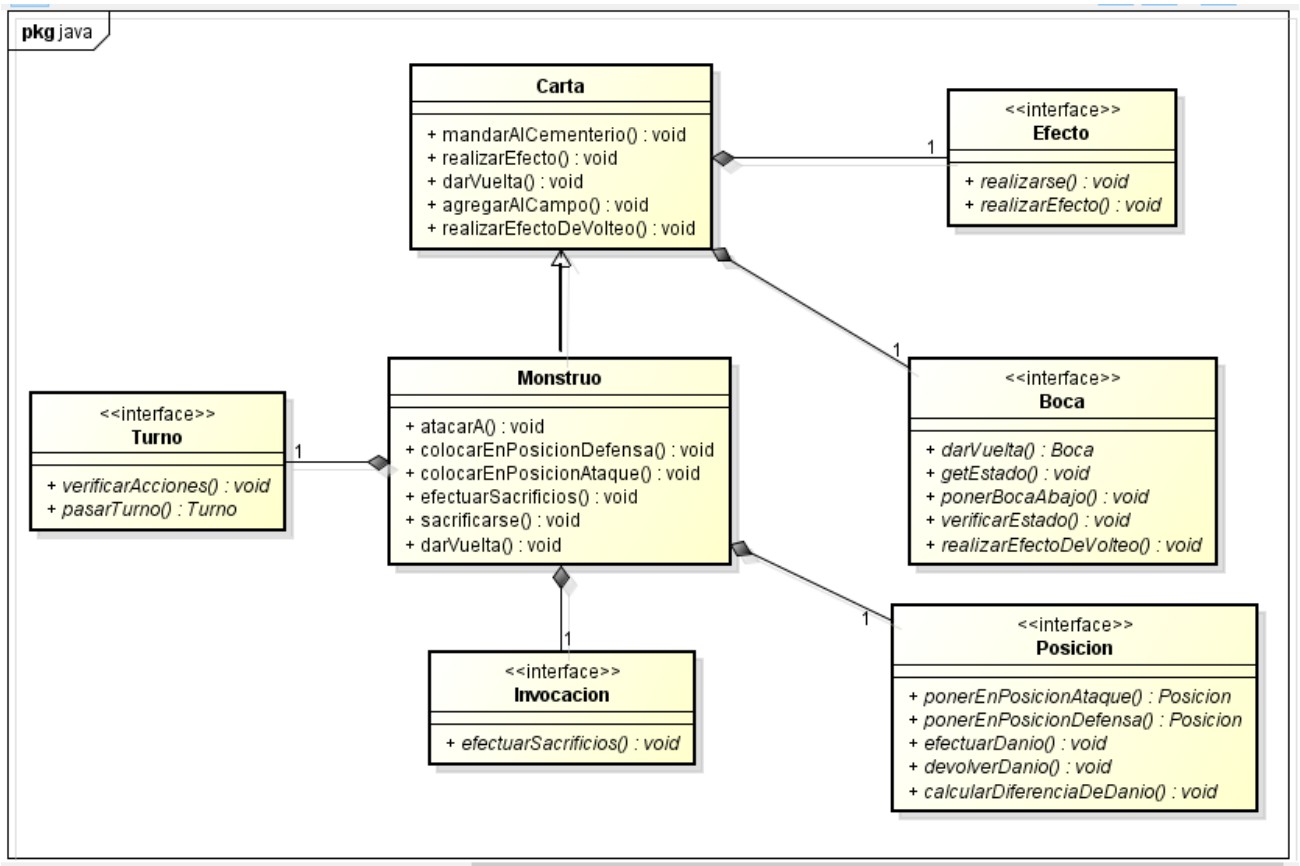

Figura 2: Diagrama de Clase de Carta.

En este diagrama se puede ver más en detalle en qué clases delega Carta cuando le envían determinados mensajes.

En primer lugar, para la realización de efectos las clases Carta y Monstruo delegan en la interfaz Efecto, cuya implementación irá variando. Esta clase Efecto es capaz a la hora de actuar (en la mayoría de los casos) vuelve a delegar en la clase Campo.

En el caso en que se efectúe un efecto de volteo, el procedimiento es un poco distinto puesto que es necesario verificar que la carta se encuentre boca abajo. Para ello, se delega sobre la interfaz Boca quien recibe el efecto del monstruo y lo activará dependiendo de la implementación.

Para los ataques en el caso de Monstruo, para el cálculo de daño se ayuda de la clase Posición. Para así poder encapsular todo el comportamiento del ataque y la defensa del monstruo en esta interfaz puesto que, según las reglas, el daño neto realizado en un ataque depende exclusivamente de la posición en que se encuentra la carta. Una vez que Posición devuelve el resultado de daño,

Monstruo hará los cambios que sean necesarios. Sin embargo, antes de realizarse un ataque, Monstruo delegerá en Boca la verificación de la validez del ataque, puesto que un Monstruo no puede atacar si se encuentra boca abajo.

Por otra parte, la clase Monstruo, antes de realizar detereminadas acciones, como dar vuelta o cambiar de posición, verifica si las puede llevar a cabo mediante la interfaz Turno.

Por último, en lo que a Invocaciones se refiere, cuando se le pide a un Monstruo que realice sacrificios, Monstruo delega en la clase Invocación. Esta última, volverá a delegar en Campo esecificando la cantidad de sacrificios necesarios de acuerdo a su implementación.

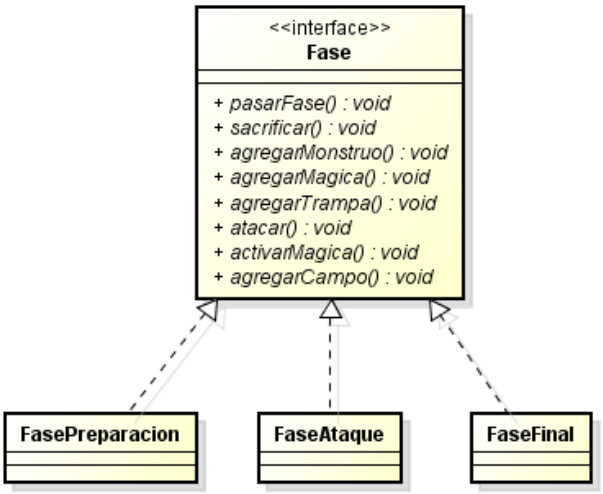

Figura 3: Diagrama de Clase de implementación de la interfaz Fase.

Aquí se muestran las tres fases del juego que implementan a la *interface* Fase. Como se dijo en la sección 3, cada una implementa los métodos correspondientes de una forma distinta según el juego en cuestión.

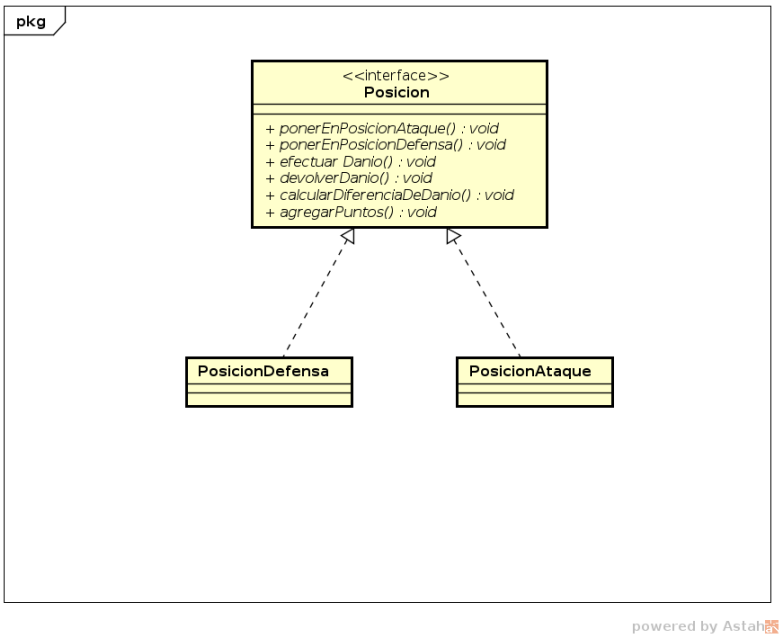

Figura 4: Diagrama de Clase de implementación de la interfaz Posicion.

En este diagrama podemos ver todas las clases que implementan Posicion. Sobre estas clases, Monstruo delegará la responsabilidad de la realización de daño y defensa de él mismo. Estas clases diferirán en cuanto a la implementación de los distintos métodos de acuerdo a las reglas del juego.

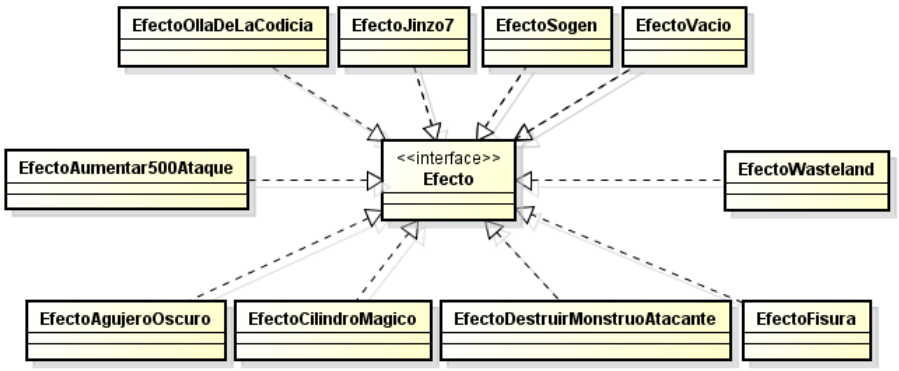

Figura 5: Diagrama de Clase de implementación de la interfaz Efecto.

En dicho diagrama muestra en donde se implementa la *interface* Efecto. Se puede ver que los diversos efectos corresponden a una respectiva carta, como se explica en el sección 3. En el caso de una carta Monstruo que no tiene ningún efecto, se le asigna EfectoVacio, que como su nombre lo indica, no realiza ninguna acción.

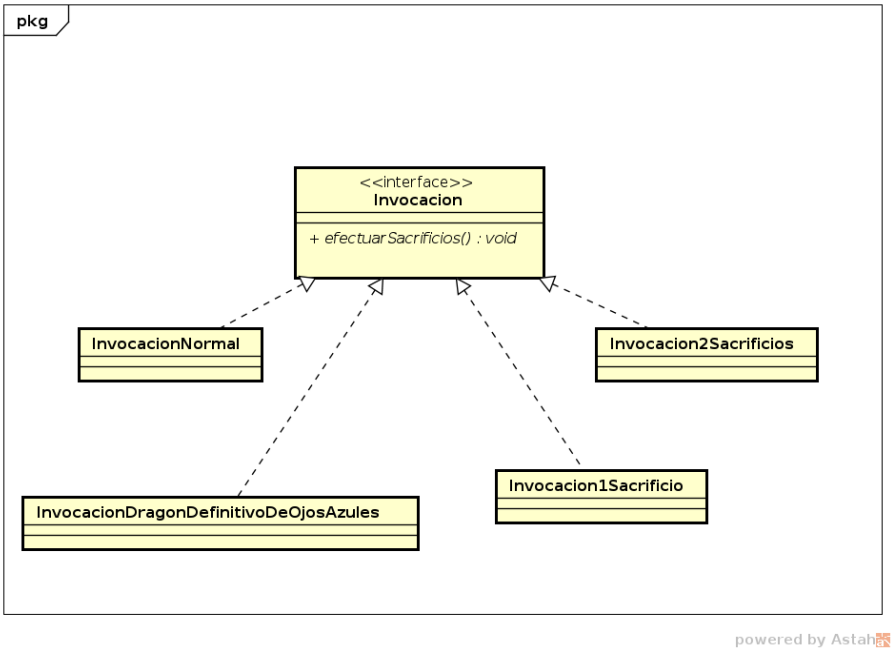

Figura 6: Diagrama de Clase de implementación de la interfaz Invocacion.

En este diagrama se pueden ver todas las clases que implementan la interfaz de Invocacion. Todas las invocaciones implementan el método de acuerdo a las reglas del juego, exceptuando InvocacionNormal cuyo método no hace nada puesto que dicha invocación no presenta requisito alguno.

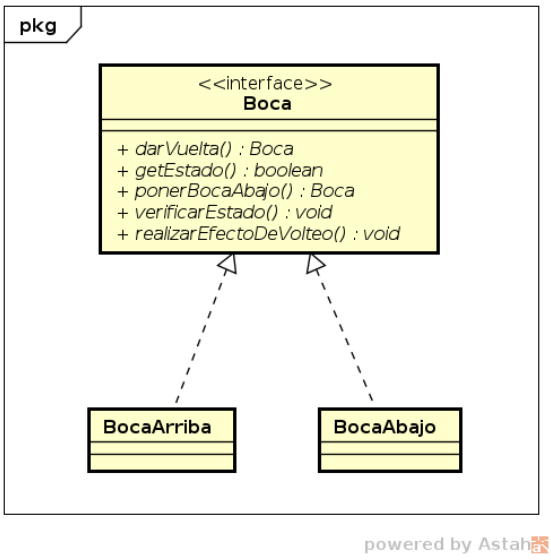

Figura 7: Diagrama de Clase de implementación de la interfaz Boca.

Acá podemos ver la implementación de Boca. Muy sencilla pero nos permite evitar de forma elegante la pregunta sobre el estado de la Carta antes de realizar ataques o efectos de volteo.

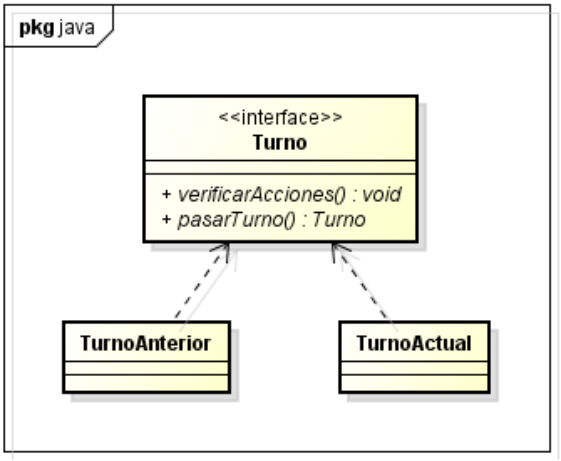

Figura 8: Diagrama de Clase de implementación de la interfaz Turno.

Por último, éste diagrama muestra, de forma gráfica, las dos posibles implementaciones para la interfaz Turno. Posee un funcionamiento muy parecido al que se encuentra en el diagrama de la Figura 7.

5. Diagramas de secuencia

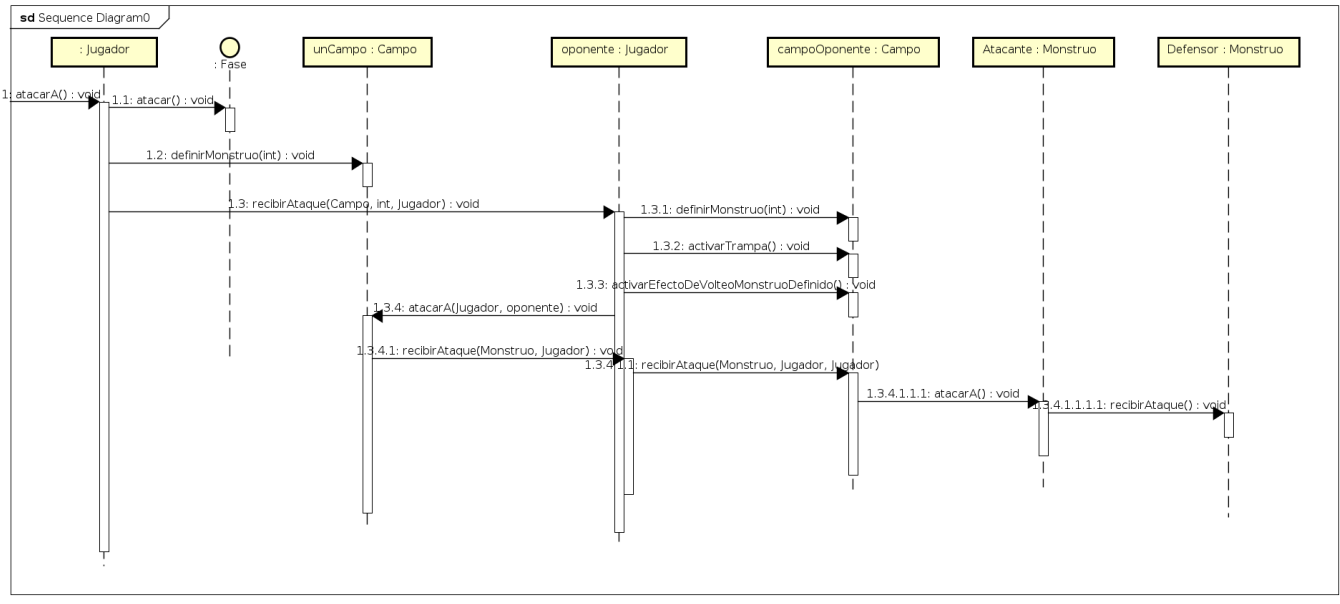

Figura 9: Diagrama de secuencia de ataque.

En este diagrama podemos ver la secuencia de mensajes que se suceden al invocarse el método de atacarA de Jugador. La idea es que primero se definan los monstruos en cada Campo, después se delegue el ataque en ellos y en última instancia en los Monstruos (ver más adelante el funcionamiento interno de los mismos).

Además el Jugador defensor tiene la posibilidad de activar trampas y efectos de monstruos, antes de seguir con el ataque.

Una vez que la secuencia llegue a los Monstruos, ellos harán las modificaciones necesarias para todos los objetos en cuestión debido a que conocen a ambos Jugadores por parámetro.

Como observación, Jugador antes de enviarle el mensaje definirMonstruo a Campo le manda un mensaje a la Fase para verificar si es posible realizar la acción en cuestión. Monstruo hace también una verificación similar pero con Boca y con Posición antes de enviar el mensaje al Monstruo defensor, lo podemos ver en el siguiente diagrama, junto con todo su funcionamiento interno de cálculo de daño:

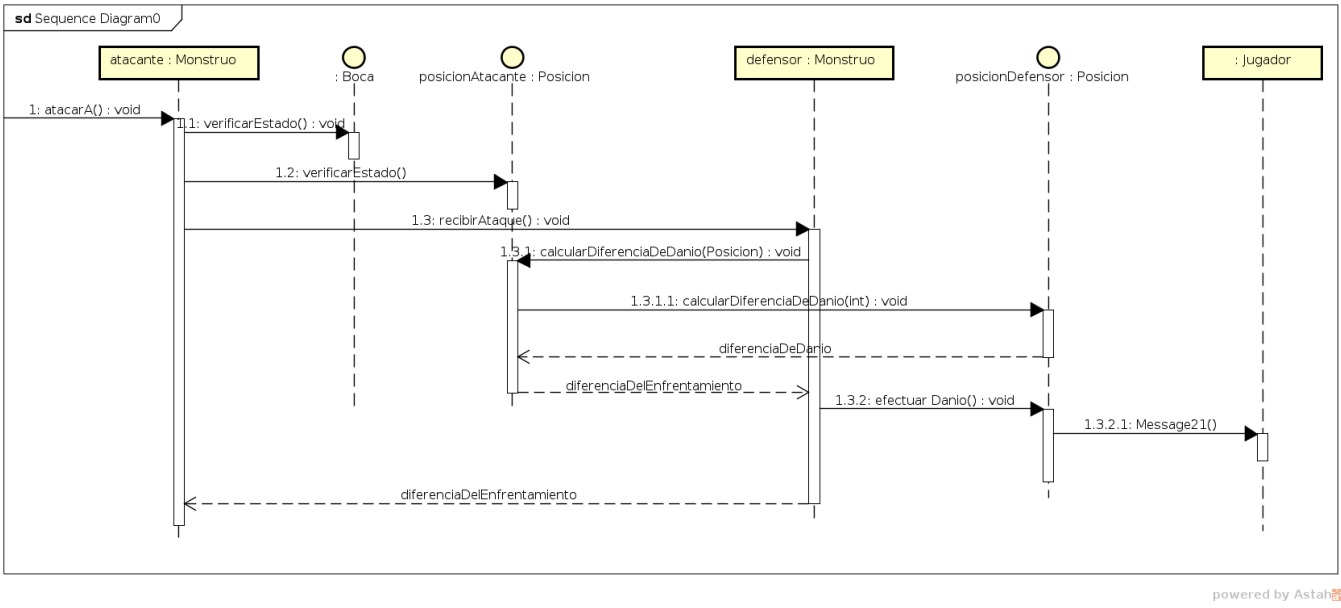

Figura 10: Diagrama de Secuencia del proceso de secuencia de ataques entre Monstruos.

Este es todo el proceso interno que se realiza entre los Monstruos una vez que se llama a atacarA.

En primer lugar, el monstruo atacante verifica si es posible realizar dicho ataque con sus estados Boca y Posicion. Después delega en el monstruo defensor quien recibe la posición del atacante. Este monstruo defensor delegará el cálculo de daño en su clase Posicion.

En este momento, la primera Posición delega en la otra pasándole sus puntos y la última finalmente realiza el cálculo de daño que se devolverá a los Monstruos. Según este resultado, los Monstruos vuelven a delegar en Posición quién realizará los cambios que sean necesarios a los Jugadores (en este caso, la Posición del Monstruo defensor tuvo que realizar un cambio en su Jugador).

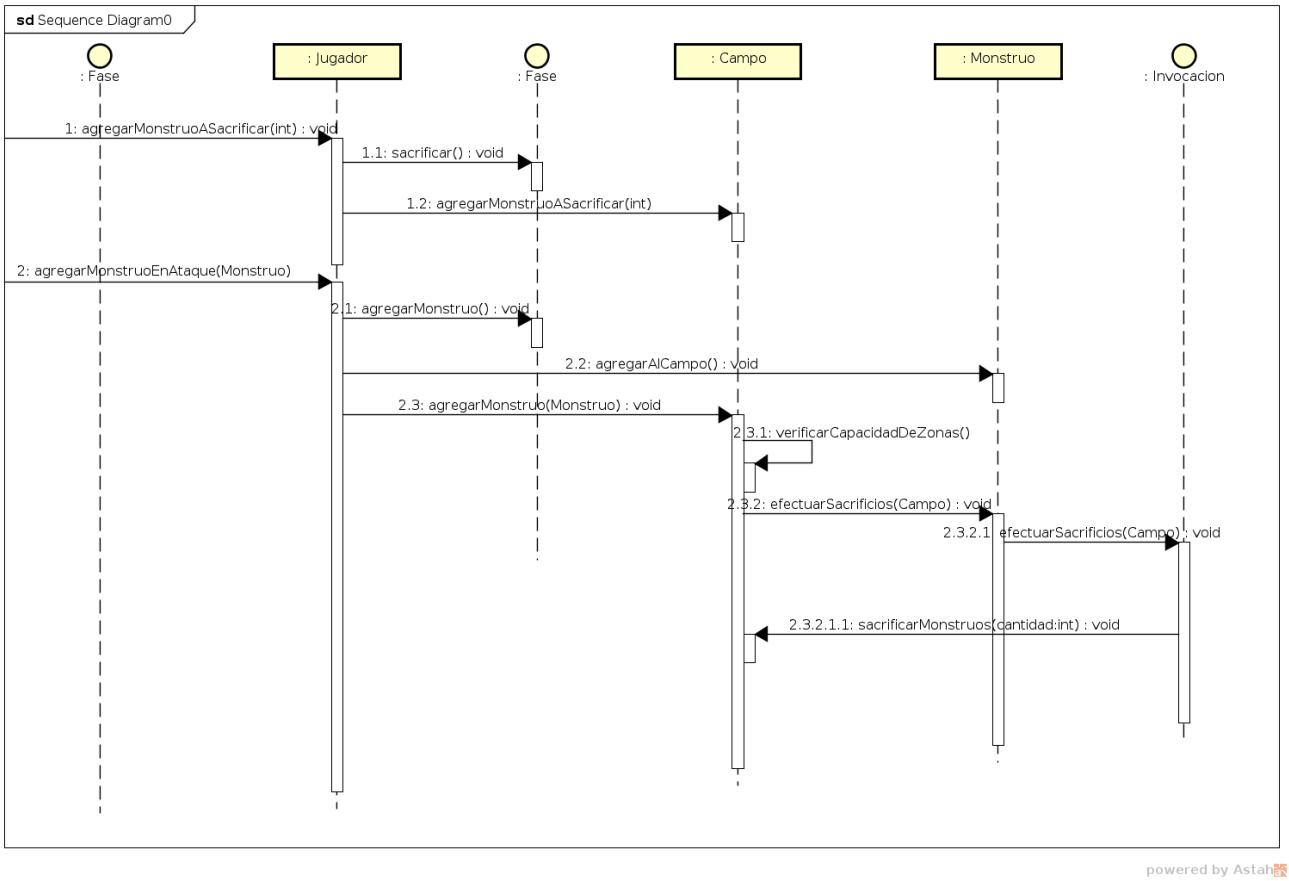

Figura 11: Diagrama de Secuencia del proceso de invocar un Monstruo.

Como podemos observar, éste diagrama de secuencia expone el proceso que ocurre cuando un Jugador quiere invocar un Monstruo. Lo primero que hace el jugador es agregar monstruos que va a ofrecer como sacrificio en su próxima invocación. Para ello le envía un mensaje a Campo, habiendo previamente verificado que se pueda realizar dicha acción en la Fase actual.

Luego, agrega el monstruo que desea al Campo (nuevamente, verificando antes con Fase). Campo antes de agregarlo, verifica que haya espacio y se asegura que cumpla los requisitos de invocación llamando al método del propio Monstruo efectuarSacrificios y pasándose como parámetro a sí mismo. Monstruo a su vez, delega en su Invocación, quien dependiendo de su implementación llama al método correspondiente de Campo para que realice los sacrificios.

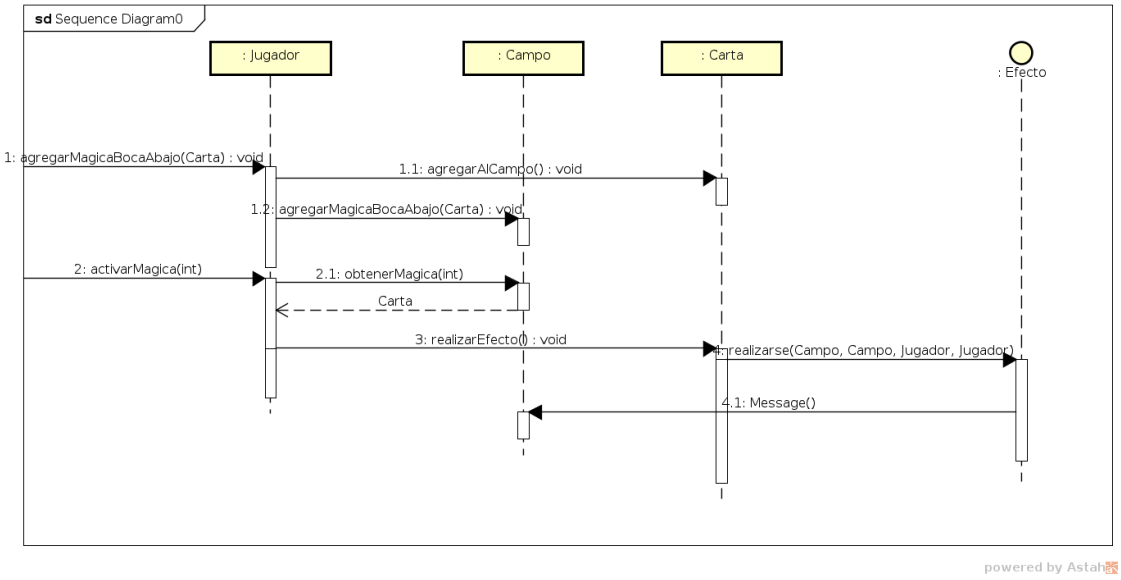

Figura 12: Diagrama de Secuencia del proceso de un efecto de la carta Magica boca abajo.

La figura 12 muestra el diagrama de secuencia del proceso en el cual una carta del tipo Magica realizar su efecto.

El primer paso lo realiza la clase Jugador agregando la carta mágica en cuestión. El Campo la agrega.

Luego, Jugador le pide la carta mágica que desea activar y Campo se la devuelve, permitiéndole a Jugador enviarle el mensaje de realizarEfecto a la carta.

A continuación, la Carta le manda el mensaje de realizarse a Efecto quién de acuerdo a qué implementación sea, mandará los mensajes que tenga que mandar los cuales en este caso, desconocemos (suponemos que le enviará alguno a Campo o puede que le envíe algo a algún Jugador también).

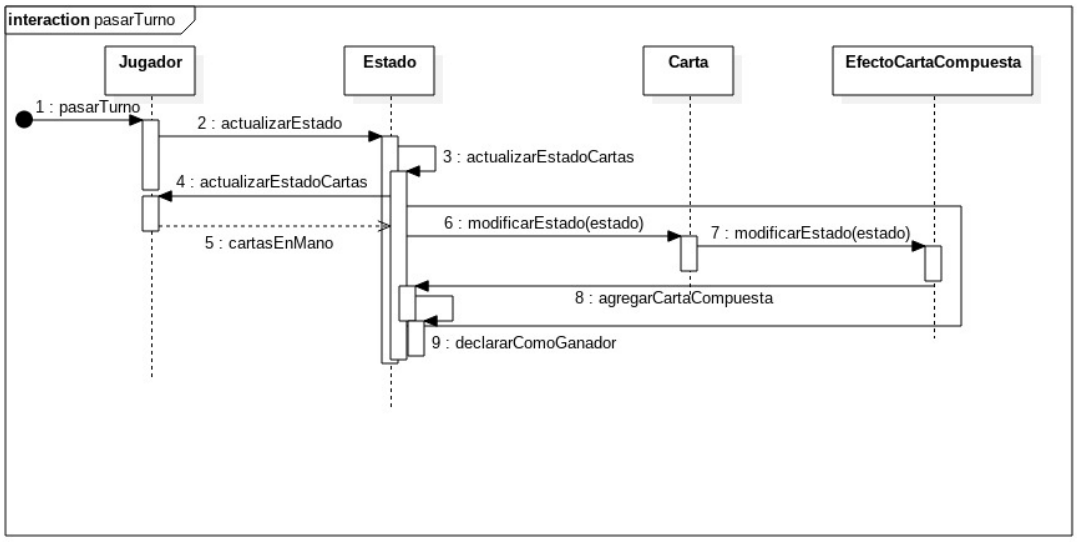

Figura 13: Diagrama de Secuencia del proceso de pasar el turno de un jugador a otro.

La imagen muestra como interactúa una instancia Jugador con su propio estado actualizándolo cada vez que se llama el método de pasarTurno en la situación de que Jugador tenga las 5 cartas de Exodia. El estado lo que hace pedirle las cartas en mano al jugador e iterarlas de manera que llama al método de modificarEstado de cada carta. Carta a su vez delega el comportamiento a su efecto, que en el caso de las cartas Exodia es el de cartaCompuesta. Este método le avisa al estado de que hay cartas de Exodia.

Como observación, Jugador antes de enviar el mensaje actualizarEstado, le manda a oponente FasePreparación (porque es la primer fase de Turno) y le hace también tomar una carta porque en caso de que el Jugador actual no gane, devuelve al oponente para que el juego siga.

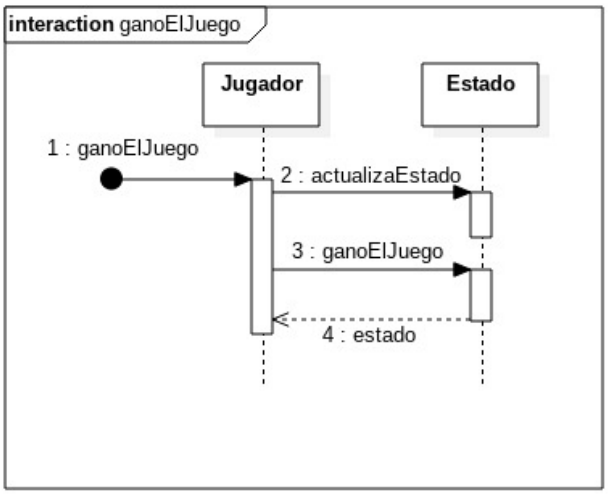

Figura 14: Diagrama de Secuencia del ver si un jugador gano.

Se ve que es una secuencia simple en la que antes de preguntarle al estado si el jugador gano, se actualiza el estado.

6. Diagramas de paquetes

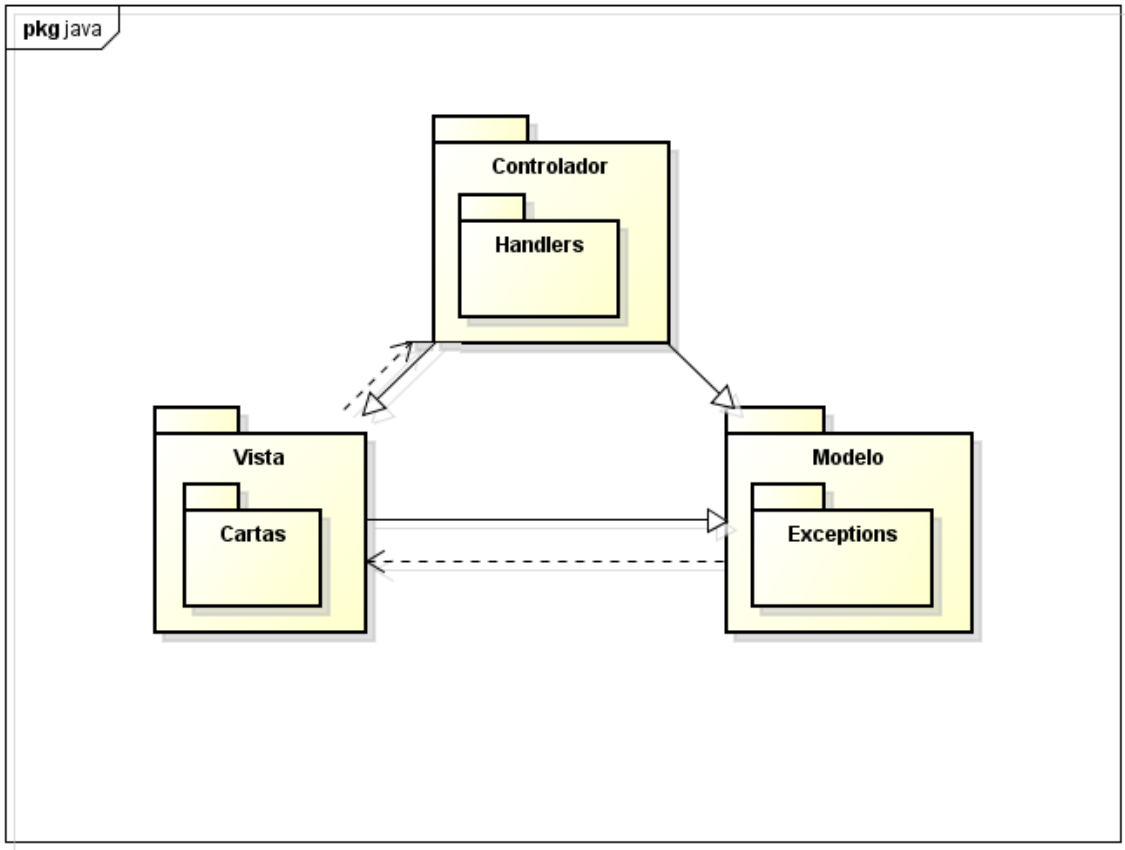

Figura 15: Diagrama de Paquetes.

7. Diagramas de estado

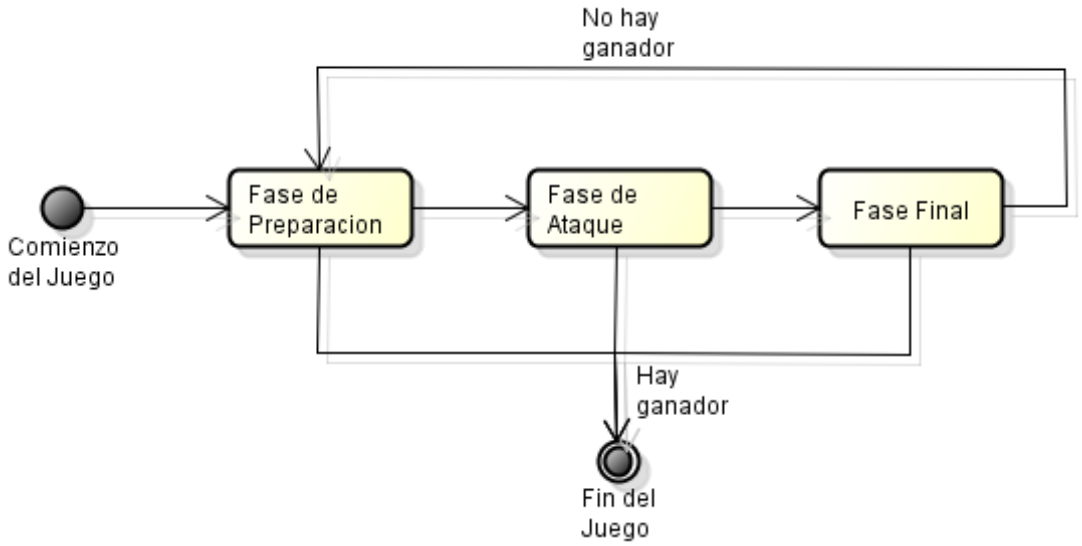

Figura 16: Diagrama de Estado de la aplicación.

En dicho diagrama podemos observar los distintos estados que ocurren desde el comienzo del juego hasta su fin. En cada turno, se comienza en la fase de preparación, y luego sigue con la secuencia indicada. Como se indicó anteriormente, en cada fase el jugador puede hacer o no, determinadas acciones.

En cada estado, antes de pasar al próximo, se verifica que ningún jugador haya ganado la partida. En éste caso, se da como finalizado el juego.

8. Detalles de implementación

Ataques

La implementación de ataques, como vimos arriba, es una gran secuencia de mensajes donde los distintos objetos van delegando responsabilidades pasando del más abarcativo al más pequeño.

La idea es que en un principio el Jugador atacante defina el Monstruo (con un mensaje para Campo) con el que va a atacar y luego le avisa al oponente que lo esta atacando a alguno de sus Monstruos, a partir de su posición en el campo. A posteriori, el oponente define cuál de todos sus Monstruos es el atacado, envíandole un mensaje al Campo.

Esta primera secuencia nos permite lograr que cada Campo 'recuerde' los monstruos que pelearán para evitar pasar constantemente, por parámetro, posiciones. Esta hecho de esta manera para que ningún Jugador 'pase por arriba al otro' sino que cada uno en base al mensaje que recibe, le avisa a su propio Campo. Esto además facilita la realización de ciertos efectos por parte de las trampas y monstruos, puesto que suelen operar sobre los Monstruos definidos.

Lo otro a destacar sobre esta implementación, son las clases encargadas de realizar el cálculo de daño (PosicionAtaque y PosicionDefensa). Estas clases fueron creadas debido a que los distintos Monstruos pueden colocarse en distintas posiciones y ello modifica las condiciones de enfrentamiento (puntos de poder, diferencia de daño que va al jugador, etc). Así, modelamos el cambio de posición con un patrón Strategy, para cambiar el comportamiento del Monstruo en lo que a las batallas se respecta.

Algo que quizás resulta un poco extraño, es que a pesar de que Jugador delega en las distintas Clases y en última instancia, en Monstruo (y por consiguiente, Posición), sigue apareciendo como parámetro de todos los métodos. La razón de esto es porque en muchos enfrentamientos es necesario modificarle los puntos de vida a alguno de los dos Jugadores y/o mandar al cementerio a alguno de los Monstruos, según sea el resultado de la batalla (entre otras cosas). Para resolver esto, decidimos justamente que se pase como parámetro a Jugador para que solamente los Monstruos y las Posiciones sean los únicos conocedores de la diferencia de números de la batalla y ellos hagan las modificaciones necesarias a los distinto objetos.

La otra opción a esta problemática, era que se calcule la diferencia y se vaya pasando el resultado para arriba y que cada Objeto haga las modificaciones que correspondan. Esto era un problema porque resultaba en una abundancia de ifs que eran necesarios para saber quién había ganado y algunas otras cosas. Por eso fue que optamos por la solución previamente descripta.

Efectos

En cuanto al modelado de los efectos, lo que hicimos fue crear una interfaz Efecto y para cada efecto nuevo pedido, creábamos una clase nueva que implemente la interfaz. De esta manera, encapsulamos la ejecución de los efectos para la Carta (puede también ser un Monstruo). Esta última, simplemente tiene que llamar al método necesario de la interfaz para activar el Efecto.

Con este modelo, las implementaciones de los efectos están completamente separados de las cartas y no tienen 'dueño', es decir, cualquier Carta podría tener el mismo Efecto que cualquier otra distinta y no habría ningún inconveniente (si bien los nombres de la clases de Efecto suelen tener el nombre de la Carta, tranquilamente podría asociar cualquier Carta con cualquier Efecto).

A la hora de realizar un Efecto, la Carta le pasa los dos Campos y los dos Jugadores para que haga los cambios que sean necesarios de acuerdo a su implementación.

Debido a este encapsulamiento, fue necesaria la creación de la clase EfectoVacío que realmente no hace nada, pero para distintas secuencias de mensajes que tienen automatizado el llamado a implementaciones de Efectos, es necesario.Por ejemplo: La secuencia de ataque, donde en un momento se llama a realizarEfectoDeVolteoDeMonstruoDefinido. En este caso, por más que el Monstruo no tenga efecto alguno (conceptualmente hablando) va seguir activando su efecto de volteo porque el Monstruo no sabe que efecto tiene. Por ello, es necesario este EfectoVacio.

Invocaciones

El modelado de las invocaciones es muy similar al de los efectos. Tenemos la interfaz de Invocacion, donde cada Invocación distinta que hay, la implementará. Como consecuencia, cada vez que se quiera invocar un Monstruo, éste delegará la realización de sacrificios en su Invocación.

A la hora de implementar las distintas Invocaciones, surgió el problema de cómo debían realizarse. En un principio, Monstruo le pasaba una lista de Monstruos a Invocación quien quitaba los necesarios para cumplir los requisitos. Esta resolución si bien funcionaba, era una violación del encapsulamiento puesto que implicaba, en definitiva que la forma de almacenar Monstruos por parte de Campo sea en forma de Lista.

Por esta razón, terminamos definiendo que cuando Campo le dice a Monstruo que realice los sacrificios, se pase por parámetro a sí mismo. De esta manera, cuando Monstruo delegue, Invocación pueda volver a delegar en Campo la eliminación de Monstruos, donde cada implementación especifica la cantidad.

Por como es la cadena de mensajes de esta interfaz, fue necesaria la creación de la clase InvocaciónNormal que en definitiva no hace nada, pero nos permite encapsular de la manera previamente descripta (similar a EfectoVacio).

Victoria por partes de Exodia

En un principio, para resolver este problema habíamos implementado un método en Monstruo que devolviera un booleano de acuerdo a si esa Carta era una parte de Exodia o no (comparando el string del nombre de la Carta con los de las distintas partes de Exodia). Este método, se llamaba cada que vez que el Jugador tomaba una Carta nueva del mazo y si lo era, se sumaba 1 en un contador interno que tenía. Cuando este contador llegaba a 5, el Jugador se declaraba como ganador. De más esta decir que esta solución no era muy linda.

Lo que surgió, fue considerar que estas partes de Exodia tuvieran alguna clase de Efecto que hiciese algo. La cuestión era qué y cómo.

Decidimos entonces crear la clase Estado. Esta clase es la responsable de decidir si un Jugador ganó o no, por ende, tendrá como responsabilidad llevar la cuenta de cuantas partes de Exodia hay en la mano. La forma en que lo hace es recorriendo la mano del Jugador cuando finaliza el turno del mismo, y a cada Carta le dice que lo modifique (a Estado) y se pasa como parámetro. A continuación, Carta vuelve a delegar en su Efecto quien dependiendo de cual sea, hará modificaciones o no. Estas modificaciones son gracias al método agregarComponenteCartaCompuesta, el cual simplemente suma 1 en el contador de Estado (si llega a 5 declara como ganador al Jugador).

De esta manera, se logra eliminar el contador del Jugador poniéndolo en una clase más acorde y además, eliminar la comparación entre nombres.

Handlers en el paquete modelo

Como se comento al principio del enunciado, las cartas al momento de crearse reciben por parámetro un clase que implemente la interfaz "HandlerCarta". Su objetivo es que cuando el controlodor interactué con las cortas, este le pedirá a cada una su correspondiente handler con el fin de poder crear un boton, de acuerdo al tipo de carta que sea. De esta manera diferenciamos las cartas Magicas, Trampas y de Campo. Este es el único aspecto por el cual valía la pena hacer una distinción de tipo. Consideramos que era conveniente hacer uso de estas interfaz y de las respectivas clases que lo implementan, ya que otra alternativa era que Mágicas, Trampas y de Campo heredasen de Carta, pero esto implicaría una gran refactorizacion por un solo método. Además es mas fácil el manejo de todas las cartas con el tipo Carta, antes que el manejo de las cartas del juego con diversos tipo. La clase Monstruo al heredar de Carta (cuyos motivos son otros de porque este tipo si herade de Carta y las otras no), si convenia que sobreecribiera el método de la clase madre.

9. Excepciones

- **AccionInvalidaEnEstaFaseException**: Esta excepción es creada por las implementaciones de Fase y lanzada siempre que Jugador intente realizar una acción que no corresponde a esa fase según las reglas del juego. Esta excepción es atrapada por la interfaz gráfica, cliente de Jugador que se encargará de mostrarle el debido mensaje al usuario.
- **CartaNoEncontradaException** : ?
- **ElMazoNoTieneCartasException** : es creada por Mazo cuando se queda sin cartas y es capturada por el Jugador. En este momento, el Jugador declara como ganador al oponente debido a que se quedó sin cartas.
- **InterrumpirAtaqueException** : es creado por algunas implementaciones de Efecto cuando se realiza un efecto de volteo debido a que estos efectos interrumpen la normal secuencia de mensajes en un ataque. Por ello, esta excepción es atrapada por el Jugador atacado y una vez atrapada, Jugador no hace nada más dando por finalizado el ataque.

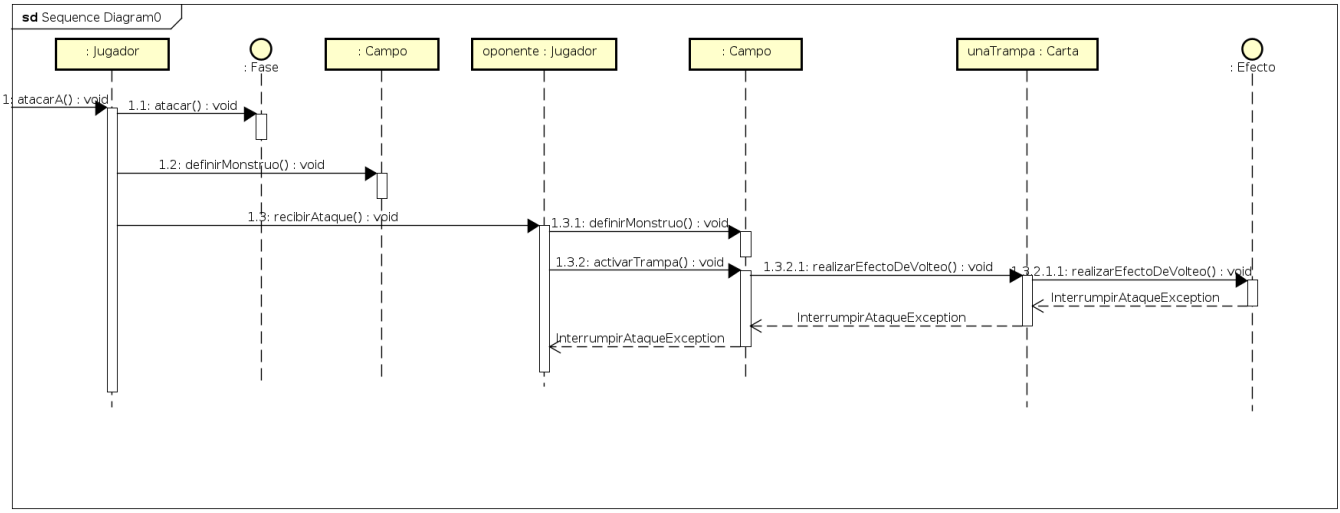

Figura 17: Diagrama de Secuencia del InterrumpirAtaqueException.

- **MonstruoNoPuedeAtacarException**: puede ser creada tanto por BocaAbajo o por PosicionDefensa cuando se intenta atacar con un Monstruo que se encuentra en alguna de esos dos estados. Esta excepción será atrapada por la interfaz gráfica para notificar al usuario del error.

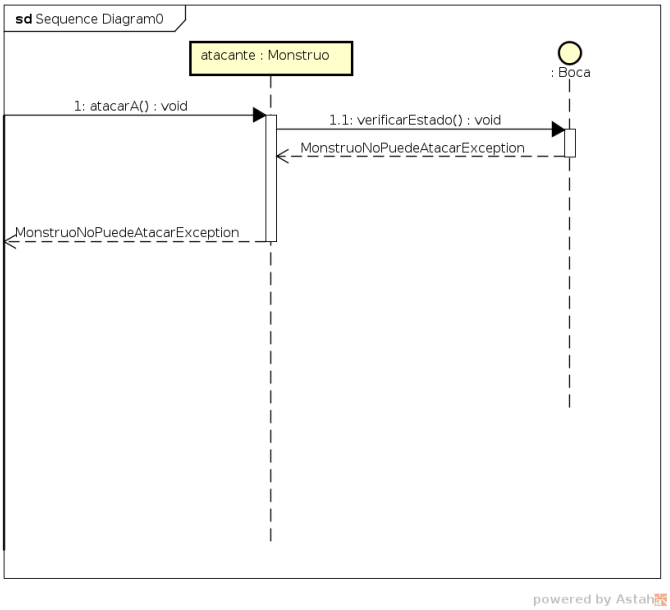

Figura 18: Diagrama de Secuencia del MonstruoNoPuedeAtacarException.

En este diagrama vemos cómo sería la creación de la excepción. La idea es que esta excepción siga de largo y sea atrapada por la interfaz gráfica.

- **MonstruosInsuficientesParaSacrificioException** : es lanzada por la clase Campo cuando el jugador actual quiere invocar un Monstruo, y no realizó, previamente, los sacrificios

necesarios para poder realizar la acción mencionada. Esta excepción será atrapada por la interfaz gráfica para que le notifique al usuario.

- **NoHayMasFasesException**: es creada por la implementación FaseFinal de la interfaz Fase, y la lanza cuando se quiere continuar acceder a una hipotética fase siguiente. Esta excepción será capturada por la interfaz gráfica y simplemente finalizará el turno del jugador.
- **ZonaNoTieneMasEspacioException**: Es creada por Campo cuando se intenta colocar una carta en alguna de las zonas y pero no hay más lugar para ella. Esta excepción es capturada por la interfaz gráfica para poder avisarle al usuario.
- **MonstruoInvocadoEnTurnoActualException**: dicha excepción es creada por la implementación TurnoActual de la interfaz Turno, y lanzada al momento en el que un jugador desea realizar una acción inválida con un Monstruo que fue invocado en el presente turno. Estas acciones son: dar vuelta la carta, y cambiar de posición (de ataque a defensa, o viceversa).