FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

[75.07] ALGORITMOS Y PROGRAMACIÓN III

1º CUATRIMESTRE 2018

TURNO NOCHE

TP2: Al-Go-Oh

AUTORES - grupo N3

Anderson, Manuel - #101.230
<manuel121097@gmail.com>

Arredondo, Nicolás - #95.618
<nicolas_arredondo@hotmail.com>

Husain, Ignacio Santiago - #90.117
<santiago.husain@gmail.com>

Parente, Gastón - #101.516
<ggparente95@gmail.com>

DOCENTES

Lic. Suarez, Pablo (tutor del TP)

Ing. Diego, Sánchez

Srta. Marijuán, Magalí

Sr. Leal Bazterrica, Matías

3 de agosto de 2018

Índice

1. Objetivo del trabajo

En el presente trabajo práctico se desarrolla una aplicación que implementa el juego de cartas Yu-Gi-Oh! utilizando el lenguaje de programación Java. Se busca cumplir los siguientes objetivos:

- realizar un análisis de la problemática planteada y su modelado mediante notación UML.
- debido a su extensión, describir resumidamente los componentes más importantes del programa.
- aplicar la teoría de programación orientada a objetos estudiada en el curso.
- utilizar la técnica de desarrollo *Test-Driven Development*.
- implementar y aplicar patrones de diseño para resolver problemas puntuales en la implementación del juego.
- realizar la interacción con los jugadores mediante una interfaz gráfica de usuario utilizando la plataforma JavaFX.
- tomar conciencia del uso del paradigma de orientación a objetos para modelar problemas que presentan una complejidad media.

2. Supuestos

Durante la programación de la aplicación y de las iteraciones en la comprensión del enunciado del trabajo, se realizaron varios supuestos que no estaban especificados en el mismo. Los mismos se basaron en las reglas del juego original [3], intentando mantener la dificultad relativa del juego, y de no complicar innecesariamente la implementación. A continuación se describe cada uno de ellos.

- Si la mano de un jugador tiene seis cartas (condición de mano llena), y el mismo toma una del mazo, entonces se va a descartar una carta de la mano del jugador. El descarte se realiza de forma automática y aleatoria, sin aviso al jugador. Dicha decisión se fundamenta en que nos pareció que le agrega dificultad al juego, ya que el usuario debe preveer este tipo de situación cuando planea diferentes estrategias de juego.
- No se puede realizar un ataque en el primer turno del juego, ya que el oponente correría con desventaja por no haber sido sorteado inicialmente para jugar.
- Cuando hay una carta campo, e ingresa una nueva, se desactiva la vieja y se la manda al cementerio.
- Se pueden activar todas las cartas mágicas que se quieran en la fase final.

- Cuando se requieren sacrificios, se selecciona automáticamente las cartas con la menor cantidad de estrellas en el campo, y se las elimina de izquierda a derecha. Además, para que las cartas se puedan sacrificar, deben estar en la región monstruo.
- Una vez utilizadas las cartas efecto, estas van al cementerio, independientemente si el efecto se efectuó. Por ejemplo, si se usa una carta Black Hole y no hay cartas en las regiones monstruo de ningún oponente, no se va a eliminar a ninguna, y la carta va a ir al cementerio. Esto simplifica la implementación, ya que no se debe verificar las condiciones particulares del juego.
- Si el jugador ataca a una carta de su oponente que se encuentra boca abajo, la misma pasa a estar boca arriba luego del cálculo de puntos de vida.

3. Modelo de dominio

Debido a que las reglas de juego son independientes de la presentación gráfica a los jugadores, y por la existencia de turnos y fases, se decidió dividir la aplicación en tres componentes que interactúan entre sí haciendo uso del patrón de arquitectura/diseño Modelo-Vista-Controlador (MVC) e implementando el patrón Observer para notificar eventos importantes entre las clases.

A continuación se describen las clases más importantes de la aplicación, junto con sus responsabilidades e interacciones con las demás.

3.1. Modelo

Basándose en los tests de integración provistos en el enunciado, se fue armando el modelo a utilizar. El mismo implementa la interfaz `ModeloInterfaz` utilizada para que el Controlador y la Vista se comuniquen con él.

La misma se encarga de la creación de los dos jugadores y de realizar las diferentes subscripciones a los eventos que suceden dentro de él, como por ejemplo, la toma de cartas del mazo, o la disminución de puntos de vida de algún jugador.

3.1.1. Jugador

La clase `Jugador` es utilizada como contenedor de sus regiones, mazo, y mano, y responsable de notificar eventos de fin de juego y de cambio de puntos de vida, y también de realizar correctamente las diferentes secuencias de juego posibles. Por ejemplo, si se le solicita que se elimine alguna carta de su región monstruo, el mismo es el responsable de quitarla de dicha región y enviarla al cementerio.

3.1.2. Regiones de juego

Se decidió crear una clase abtracta `Region`, utilizada principalmente para contener las diferentes cartas que se van jugando en la partida. De la misma derivan las clases `RegionCampo`, `RegionCementerio`, `RegionMagicasYTrampas`, y `RegionMonstruo`, y define dos métodos

importantes `colocarCarta()` y `removerCarta()`, que no solo se encargan de verificar la correcta agregación o remoción de cartas en la región, si no que también notifican a sus observadores de dichos eventos, como se explicará en una sección posterior.

Haciendo uso de la redefinición de métodos, `RegionCampo` activa o desactiva el efecto que produce la carta campo activa sobre las cartas que ingresan o egresan de las regiones monstruo del jugador y su oponente.

3.1.3. Cartas

Para modelar las cartas del juego, se decidió crear la clase abstracta `Carta` que define algunos métodos básicos que toda carta debe tener, como por ejemplo, saber quién es su jugador propietario. Se definieron cuatro clases abstractas de carta que implementan los comportamientos particulares definidos en el enunciado y que heredan de la clase madre `Carta`. Estas son:

- `CartaCampo`: define métodos abstractos para aplicar efectos y deshacerlos, y define métodos para modificar y restaurar los puntos de ataque y defensa de las cartas monstruo a las cuales afecta.
- `CartaMagica`: mediante el método abstracto `efectoParticular()` obliga a las diferentes cartas que hereden de ella a definir el efecto que provocan en el juego cuando este tipo de cartas son activadas. Además, tiene la responsabilidad de destruir la carta una vez que su efecto se utiliza.
- `CartaMonstruo`: modela las cartas monstruo definidas en el enunciado, proveyendo al cliente de diferentes métodos para calcular y modificar puntos de ataque y defensa. La misma tiene dos responsabilidades importantes. La primera es la de proveer un marco de definición de ataques, donde las cartas que heredan de esta clase deben definir la estrategia de ataque especificándola mediante la interfaz `EstrategiaAtaque`. Además, define cómo recibe los ataques de otra carta, implementando las reglas definidas en los tests de integración. La segunda responsabilidad importante que tiene es la de realizar una correcta invocación, ya que hay cartas que requieren sacrificios, por lo que implementa la estrategia de sacrificios definida en los supuestos con los que se trabaja.
- `CartaTrampa`: similar a las cartas campo, se encarga de definir los efectos que producen cuando se las utiliza en la fase trampa.

3.1.4. Orientación, Modo, y patrón State

Debido a que todas las cartas pueden encontrarse con orientación arriba o abajo, se creó la interfaz `Orientacion` para implementar qué sucede cuando hay cambios de orientación en las cartas, y para consultar la orientación de las mismas. Mediante la implementación del método `cambiarOrientacion()`, las diferentes cartas pueden especificar diferentes efectos especiales, como por ejemplo la carta Come Hombre, que si cambia de orientación cuando una carta la ataca, la misma destruye la carta que la atacó.

Las cartas monstruo puede estar en dos modos diferentes; ataque o defensa, y el mismo define qué puntos se utilizan para realizar la estrategia de ataque. Es por ello que se definió

la interfaz `Modo`, implementada por las clases `ModoAtaque` y `ModoDefensa`, que básicamente hacen uso del patrón State para definir el modo en el que se encuentra cada carta; cuando hay un pedido de cambio de modo, se asigna alguna de las dos clases a la carta.

3.1.5. Mano

Tiene la responsabilidad de actuar como contenedor de las cartas en la mano de los jugadores, y de verificar si se cumple la condición de fin de juego de tener las cinco partes de Exodia en ella. Cada vez que ingresa una carta a la mano, la misma verifica si hay partes de Exodia presentes, y cuántas hay. Si se completan las cinco partes, da aviso de dicha situación a sus observadores, implementando la interfaz `FinDeJuegoObservable`, como se explica en una sección posterior.

3.1.6. Mazo y patrón Factory

Para la creación de cartas, se utilizó la clase `FabricaCartas` que contiene cuatro fábricas de las diferentes cartas presentes en el juego. La decisión de hacer esto fue para que el cliente no se exponga a los detalles de creación de las cartas, por lo que se implementó el patrón **Factory** para implementar dichas clases. Una de las reponsabilidades que tiene es la de asignar correctamente al jugador dueño y oponente de cada carta que crea, ya que estos son utilizados luego para implementar diferentes ataque y efectos.

La clase `Mazo` se creó para contener las cartas de los jugadores, y la misma tiene la responsabilidad de crearlas cuando comienza el juego. Además, las mezcla aleatoriamente y mantiene consistente el estado del mazo, no permitiendo tomar más cartas de las que posee.

Por otro lado, de la misma forma que la mano, es la responsable de verificar la condición de fin de juego de si el jugador se quedó sin cartas en el mazo, y de avisarle a los observadores de fin de juego que se cumplió dicha condición para que se termine la partida.

3.2. Controlador

La clase `Controlador` tiene la responsabilidad de implementar las reglas que hacen referencia a turnos y fases del juego, ya que es la que permite al jugador realizar diferentes pedidos. Para hacerlo, delega algunas verificaciones en dos clases importantes:

- `MaquinaTurnos`: se encarga de sortear el jugador inicial y del control de turnos y fases. Las fases se implementan mediante la interfaz `Fase`, y hace uso del patrón State para especificar la fase en la que se encuentra el juego. Cada vez que se termina el turno del jugador, realiza un *swap* del jugador y oponente actual. Por otro lado, tiene la responsabilidad de mantener un registro de las cartas que se jugaron actualmente, si se tomó o no una carta del mazo en el turno del jugador, si se colocaron cartas en la región monstruo, y si alguna de las cartas en el campo atacó a una del oponente. La utilidad de llevar estos registros es que permite la simplificación de la verificación de las diferentes reglas del juego, ya que interactúa con la otra clase de soporte descripta a continuación.

- `VerificadorCondicionesJuego`: la utiliza el controlador para verificar si el jugador puede llevar acabo la acción que le solicita al programa, ya sea desde si puede jugar en

el turno actual, envíar monstruos a su región campo, o activar cartas mágicas. Además, define un `EstadoVerificador` para informar al cliente de si el estado de verifación fue bueno o fallido cuando le realiza las consultas de estado de juego. Las diferentes condiciones de juego se implementan en las clases dentro del paquete `condicionesJuego`.

Todos los pedidos al controlador se realizan mediante la interfaz `ControladorInterfaz`, que es utilizada por la vista para elevar los pedidos del usuario mediante la interfaz gráfica. En definitiva, el controlador fue creado para abstraernos de las reglas de juego y turnos, y para proveer una interfaz entre los pedidos de usuario que llegan desde la vista y se dirigen al modelo.

3.3. Vista

Debido a que la aplicación posee varios eventos disparados por los jugadores, se creó la clase vista para poder interactuar con ellos mediante una interfaz gráfica utilizando JavaFX como plataforma. La misma se encarga de contener una escena y realizar las actualizaciones de dibujo cuando suceden diferentes eventos.

Uno de los elementos más importantes de la vista es la interfaz `Escena`, ya que es la que define la escena que se está mostrando actualmente a los jugadores. Para separar las diferentes escenas, se crearon las clases
`EscenaBienvenida`, `EscenaFinDeJuego`, `EscenaSorteoJugadorInicial`, y
`EscenaTableroPrincipal`, donde en cada una se define el comportamiento que debe tener la interfaz gráfica dependiendo del estado del juego. Se refiere al lector a la sección sobre diagramas de estado para mayor detalle.

Todas de ellas hacen uso de objetos multimedia para reproducir videos o para ejecutar sonidos cuando el usuario realiza cierta acción, como algún ataque o uso de carta mágica.

La escena del tablero principal define varias regiones en la misma, donde se posicionan las diferentes cartas y se da información a los jugadores sobre de quién es el turno actual, en qué fase se encuentra, y sobre puntos de vida de cada uno. Una imagen de la misma se muestra en la figura 3.1.

Figura 3.1: Vista de tablero principal, donde se puede ver al jugador Harry Potter dándole una lección a Voldermort.

Se hizo uso de la clase `Tooltip` para mostrar información al usuario de las cartas, como se puede ver en la figura 3.2.

Por otro lado, para los ataques de cartas monstruo, se utilizó un menú como el que se muestra en la figura 3.3 para que el jugador seleccione la carta a atacar.

Figura 3.2: Vista de tooltip con información de carta monstruo.

Figura 3.3: Menú de ataque.

Para realizar los cambios de turno y fase, se utiliza la clase `EstadosJuegoBotones` que define los dos botones que puede utilizar el jugador para realizar dichos cambios.

Además, para mostrarle al jugador la información de la fase y turno actual, y de los puntos de vida de cada uno de ellos, se crearon las clases `TurnoActualVista` y `VidaVista`.

Por último, para notificar al jugador que no se pueden realizar ciertas acciones, se creó la clase `ErroresVista` que se encarga de mostrar diferentes advertencias. Por ejemplo, como se puede ver en la figura 3.4, el jugador "Voldemort" intentó atacar al jugador "Harry Potter" en el primer turno, produciéndose la advertencia correspondiente.

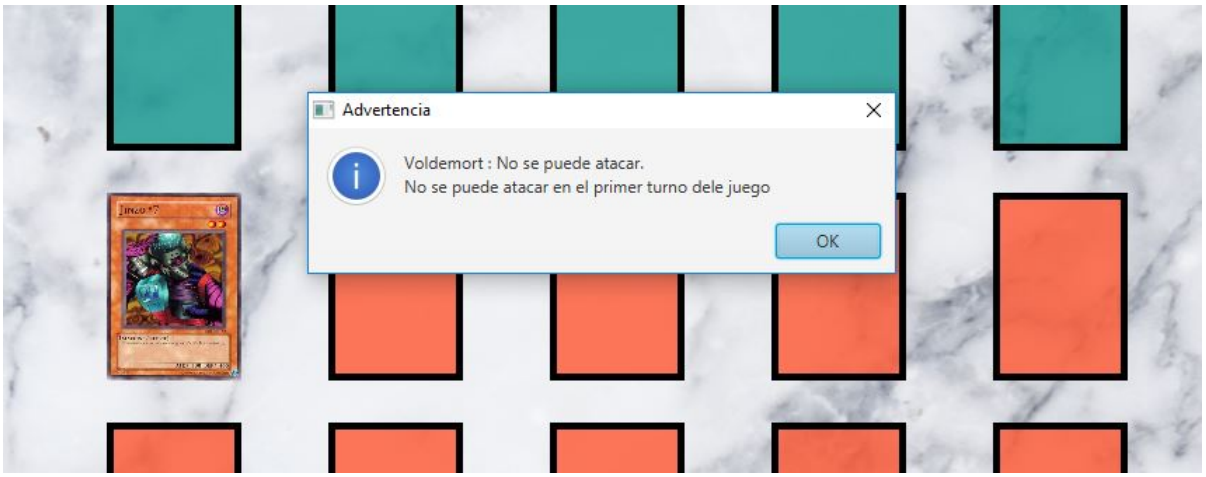

Figura 3.4: Voldemort intentando atacar a Harry Potter en el primer turno.

3.4. Notificación de eventos mediante patrón Observer

Se decidió utilizar el patrón observer para notificar todos los eventos que ocurren en la aplicación, ya sea para actualizar la escena actual en la vista, como para notificar eventos de fin de juego, o también para saber si entraron cartas a las diferentes regiones para implementar los efectos que existen en el juego.

Particularmente, la clase `Modelo` implementa diferentes observadores para escuchar eventos de interés, y las clases que observa son:

- `Carta`, y le notifica si hubo cambios de orientación en la carta.

- `Mano`, y esta le notifica si se agregó o quitó una carta de la mano de cualquiera de los jugadores.
- `Mazo`, y le notifica si se tomó una carta de alguno de los dos mazos.
- `Region`, y le notifica si ingresó o egresó una carta de alguna de las regiones.
- `Jugador`, y le notifica si se modificaron los puntos de vida del mismo, por ejemplo, durante un ataque.

Con esta información, la clase `Modelo` le notifica a la Vista para que la misma actualice su estado y le muestre al jugador el evento ocurrido.

Por otro lado, se tiene la clase `RegionCampo` que observa a las regiones monstruo del jugador y del oponente para activar o desactivar el efecto de la carta campo activa actualmente (si es que la hay) cuando ingresa o egresa una carta monstruo de dichas regiones.

3.4.1. Observador de fin de juego

Para saber si se cumple alguna de las condiciones de fin de juego, se definen las interfaces `FinDeJuegoObservable` y `ObservadorDeFinJuego` para notificar al Controlador de que se llegó al fin de juego, para que este realice las tareas necesarias de terminación. Entre otras, mostrar la pantalla de salida al usuario.

Las clases que son responsables de notificar un fin de juego son `Mazo`, `Mano`, y `Jugador`, ya que pueden darse las diferentes causas de fin de juego en ellas. Por ejemplo, si el mazo de algún jugador se queda sin cartas, va a notificar tal evento para finalizar la partida.

Las diferentes causas de fin de juego se encuentran dentro del paquete `finDeJuego`, y todas derivan de la clase abstracta `CausaFinJuego`. En la misma se define el método `getNombreCausa()` que es utilizado por la vista para mostrarle al jugador cuál fue la causa de fin de partida.

3.5. Utilización de patrón Singleton

Hay varias clases en las que se requiere una sola instancia de las mismas durante la aplicación. Por ello, se decidió utilizar el patrón Singleton para evitar que el cliente de las clases cree más de una de ellas.

Hay varias de ellas, pero las más importantes son `Controlador`, `Modelo`, las diferentes fases del juego, `MaquinaTurnos`, `VerificadorCondicionesJuego`, y las diferentes clases "nulas": `CartaTrampaNula`, `CartaMonstruoNula`, `CartaNula`, `FaseNula`, `CausaFinJuegoNula`, etc...

4. Diagramas de clases

A continuación se muestran los diagramas de clase estáticos. Debido a la extensión del trabajo, se decidió mostrar solamente las clases y métodos más importantes, por lo que se refiere al lector al código fuente para estudiar los detalles de implementación.

4.1. Carta

En la figura 4.1 se muestra como está organizada la clase Carta. Como ya se mencionó anteriormente, Carta es una clase abstracta, la cual es "padre" de todos los tipos de cartas en el juego (hablaremos de cada una de ellas más adelante). Todas las cartas son creadas en la Fábrica de Cartas, figura 4.6.

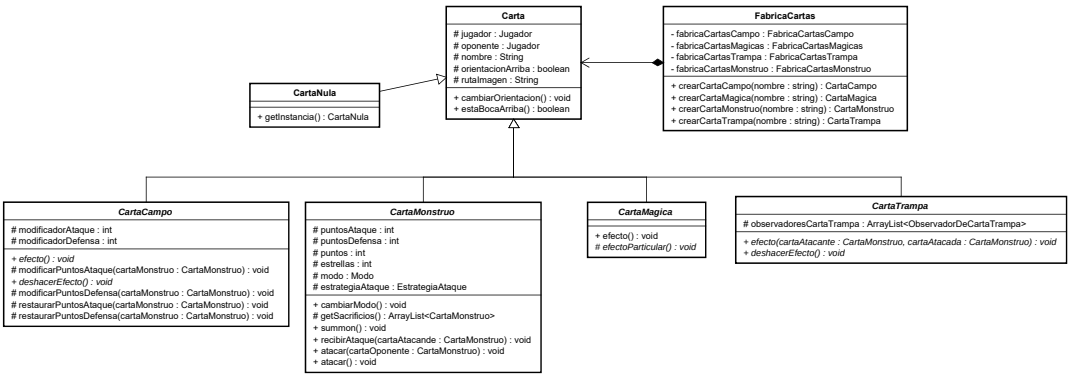

Figura 4.1: Diagrama de la clase Carta

4.2. Carta Monstruo

En la figura 4.2 notamos que este tipo de cartas tienen un `Modo` y una `EstrategiaAtaque`. El modo nos dice, como muestra la figura, si la carta esta en ataque o defensa. De esto va a depender ya sea con que puntos (Puntos de ataque o puntos de defensa) se realiza el cálculo para determinar que carta "gana" en caso de un ataque y las opciones que tiene la carta cuando ya fue jugada. Por ejemplo, una carta en modo defensa no puede atacar.

En cuanto a la `EstrategiaAtaque`, esta es la entidad que se encarga de "ejecutar" el ataque. Fue creada ya que en el juego, pueden ocurrir situaciones en las cuales un ataque no es el común, por ejemplo, como se muestra en la figura, el caso en el cual una carta `MagicCylinder`

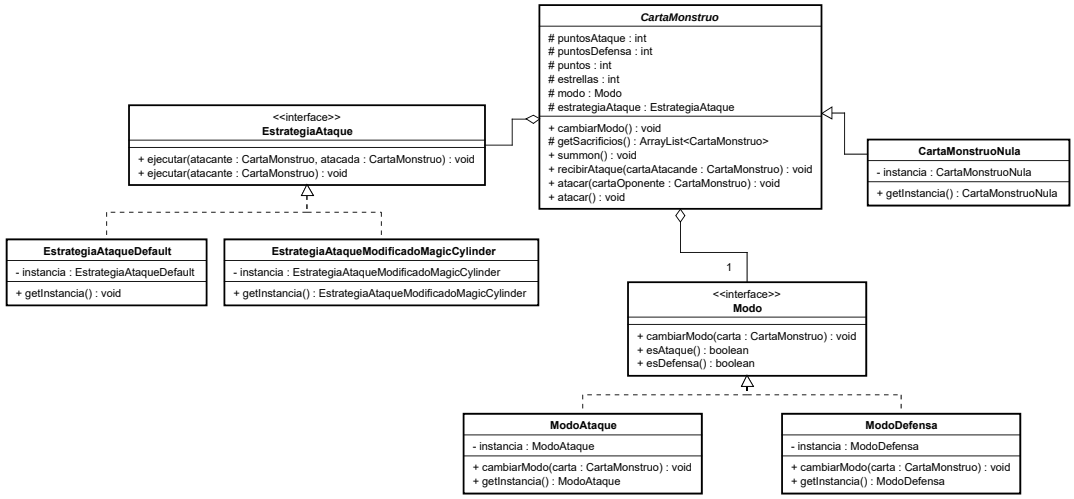

Figura 4.2: Carta Monstruo.

4.3. Carta Trampa

Refiriéndose a la figura 4.3, la característica principal que vemos es que este tipo de cartas tienen un efecto y, además, pueden influir en como se ejecuta un ataque de una `CartaMonstruo` (como ejemplificamos anteriormente con la carta `MagicCylinder`). El efecto de la carta es diferente para cada una de las `CartaTrampa`.

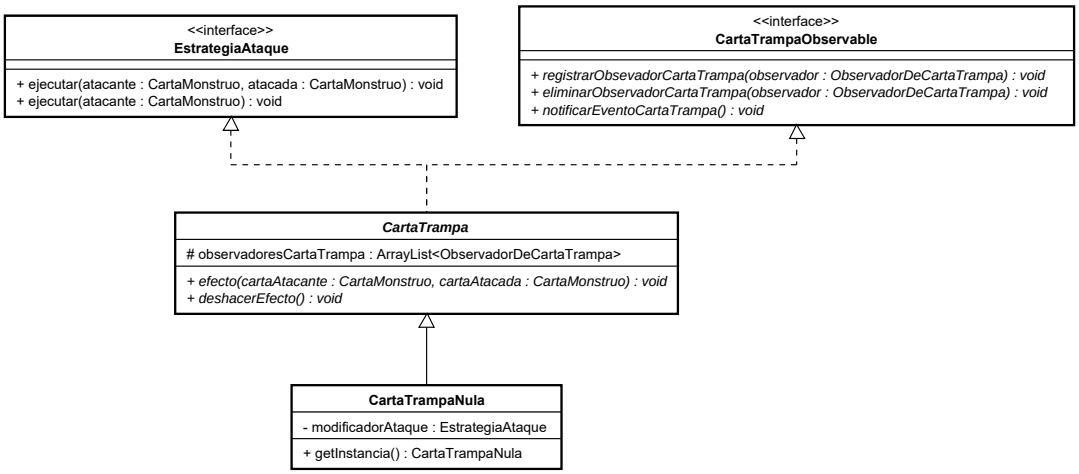

Figura 4.3: Carta Trampa

4.4. Causa Fin de Juego

Figura 4.4: Causa fin de juego. El “juego” puede terminar por diversas razones, por lo que se crearon clases para cada una de las posibilidades de fin de juego, las cuales, a medida que corre el programa, se encargan de verificar, interactuando con otros elementos del modelo, que se hayan o no cumplido las condiciones para que termine el juego.

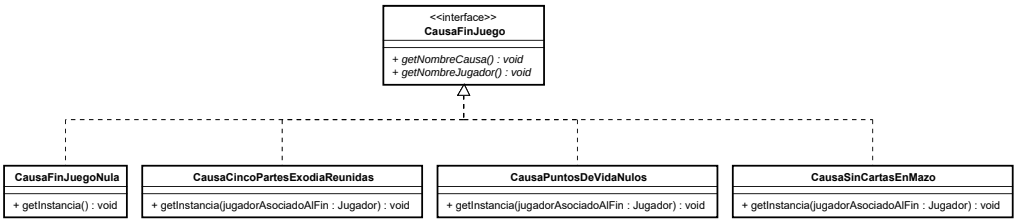

Figura 4.4: Causa Fin de Juego.

4.5. Controlador

Aquí (Figura 4.5) vemos como aparece el patrón MVC mirándolo desde el `Controlador`. Este tiene relaciones con el `Modelo` al igual que con la `Vista` y tiene dentro de el a `VerificadorCondiciones` clase que se encarga de realizar las verificaciones necesarias para habilitar o no una acción del `Jugador`.

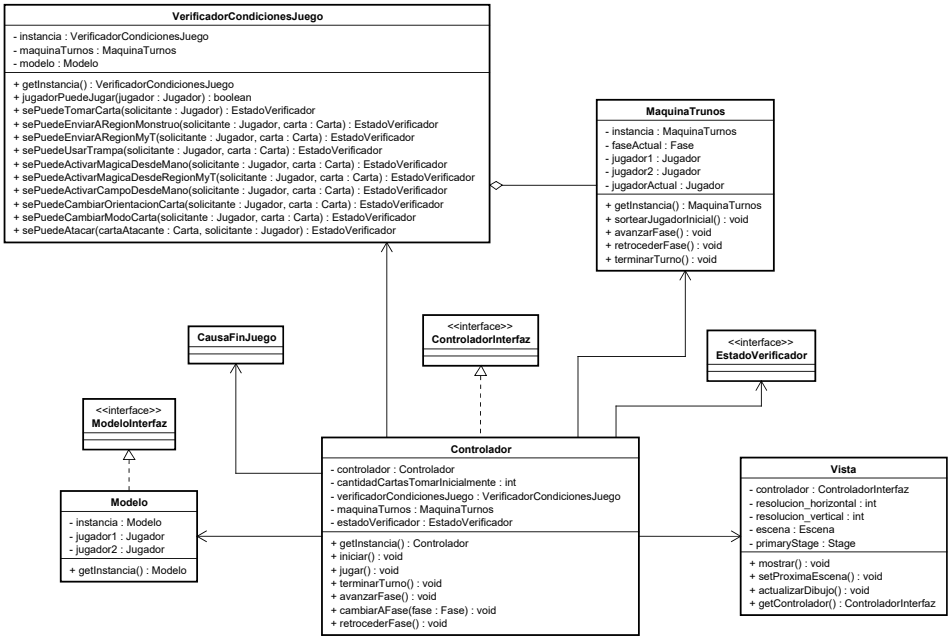

Figura 4.5: Controlador.

4.6. Fábrica de Cartas

En el diagrama (4.6) podemos ver a la `FabricaDeCartas` y sus 4 “SubFabricas”. Como se ve en la figura, tenemos una Fabrica para cada tipo de carta. Cada una de estas fabricas es creada con todas las cartas posibles del tipo especificado dentro de ella, lo que quiere decir que, todas las diferentes `CartasMonstruo` son creadas dentro de `FabricaCartasMonstruo`, luego, la `FabricaDeCartas` le pide a la `FabricaCartasMonstruo` la carta específica que quiere.

Como se menciono en la nota del diagrama, la cantidad de cartas que tenemos en cada fábrica es limitada. Pero de la forma que fueron hechas, si en algún punto queremos agregar, por ejemplo, una nueva `CartaMonstruo`, alcanza con crearla con sus respectivos puntos de ataque, defensa y efectos dentro de la `FabricaCartasMonstruo` para que pueda ser usada en el juego.

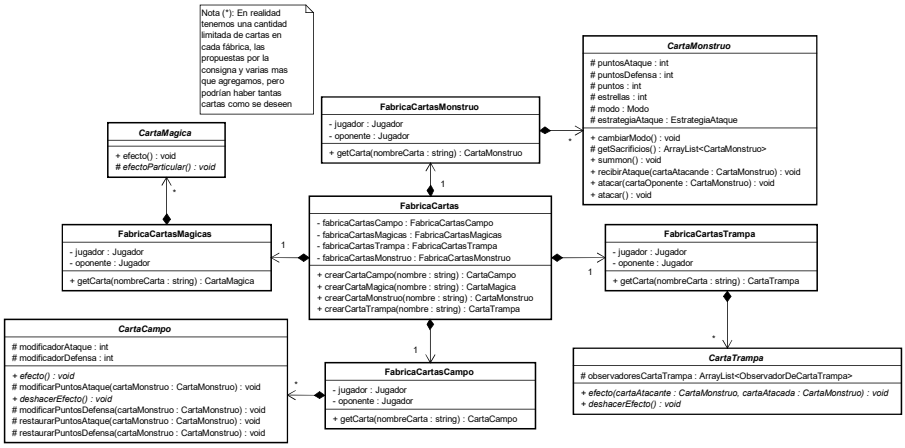

Figura 4.6: Fábrica de Cartas.

4.7. Jugador

Aquí vemos (Figura 4.7) como esta compuesto el `Jugador`. Éste, como se menciono anteriormente, esta compuesto por su oponente (otro `Jugador`) su `Mano`, `Mazo` y 4 `Regiones` (Figura 4.10).

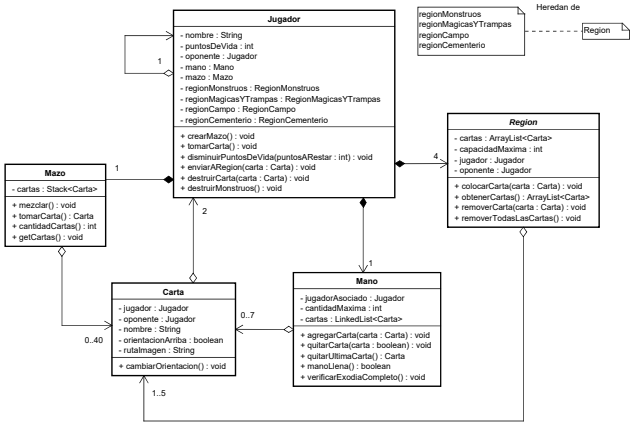

Figura 4.7: Jugador.

4.8. Modelo

El diagrama a continuación (Figura 4.8) muestra los métodos mas importantes en la interfaz `ModeloInterfaz`. Estos métodos son los que la clase `Controlador` usa para realizar cambios en el `Modelo` a partir de las acciones que realiza el usuario en la `Vista`.

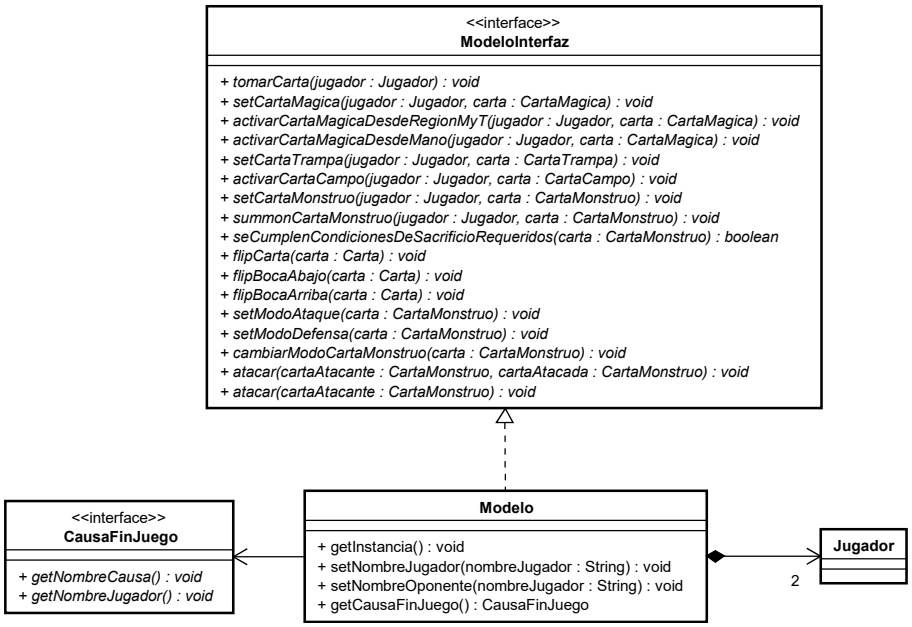

Figura 4.8: Modelo

4.9. Observadores

En el diagrama (Figura 4.9) se pueden ver todas las relaciones entre los observadores del juego, para facilitar la comprensión solo se nombraron los métodos más importantes de cada `Observador`.

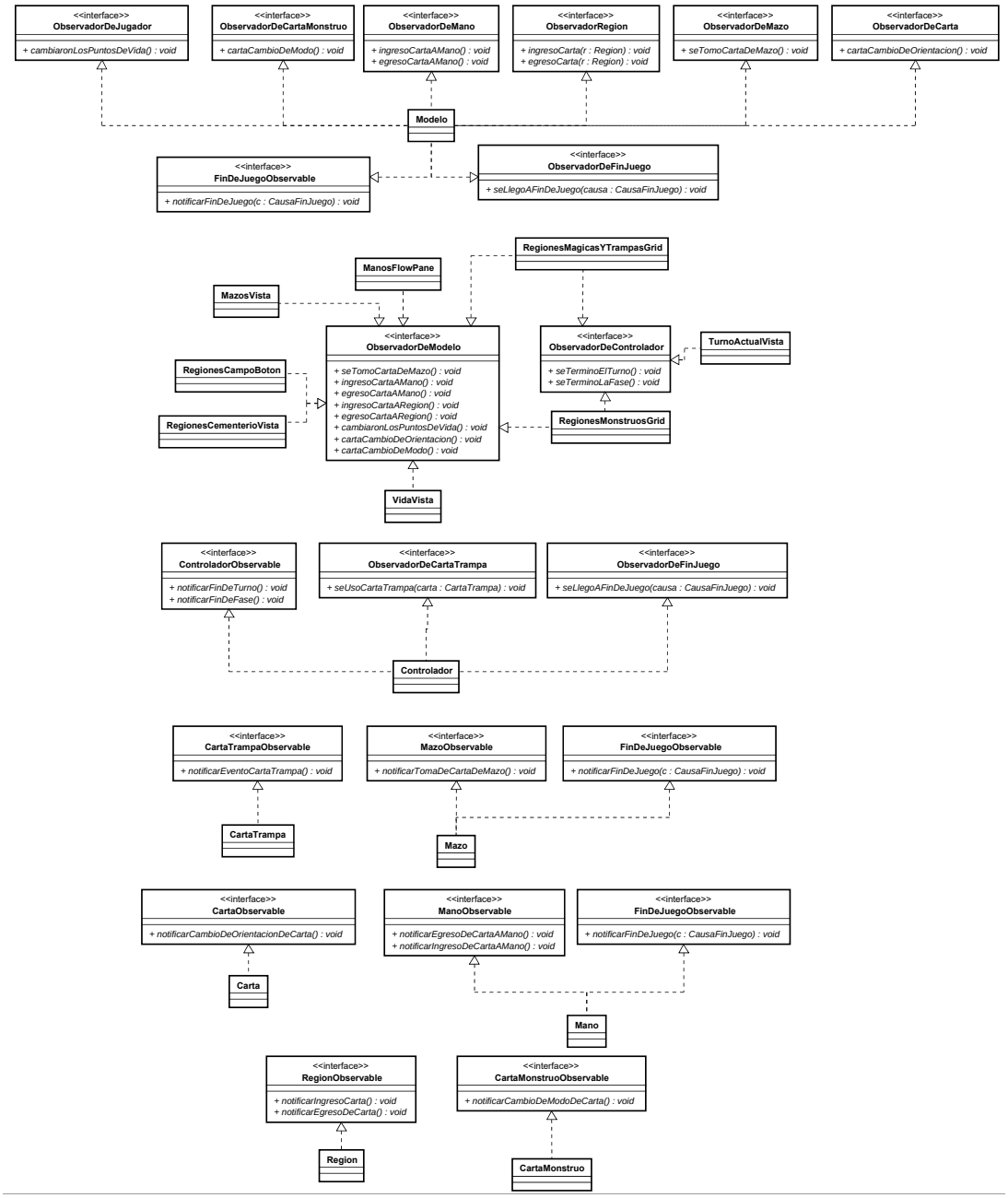

Figura 4.9: Observadores.

4.10. Regiones

La figura 4.10 muestra la relación de herencia entre los distintos tipos de regiones (`RegionCampo`, `RegionMonstruos`, `RegionCementerio` y `RegionMagicasYTrampas`) con su clase "padre" `Region`, con algunos métodos generales para todas las regiones, pertenecientes a la clase abstracta `Region` y específicos para cada una de sus diferentes clases "hijas".

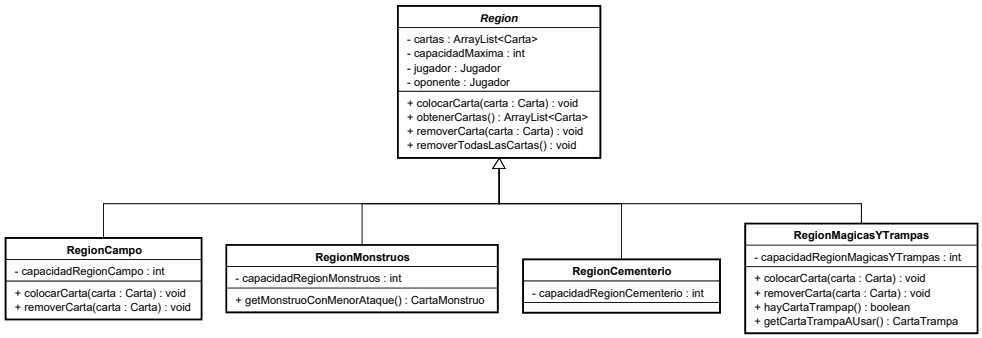

Figura 4.10: Regiones.

4.11. Vista

El diagrama (Figura 4.11) muestra las distintas variaciones que puede tener la `Escena` a través del juego (Se habla más sobre esto en la sección diagramas de secuencia, Figura 7.1), acompañado de los métodos principales que contiene la interface `Escena` y algunos específicos de cada escena específica.

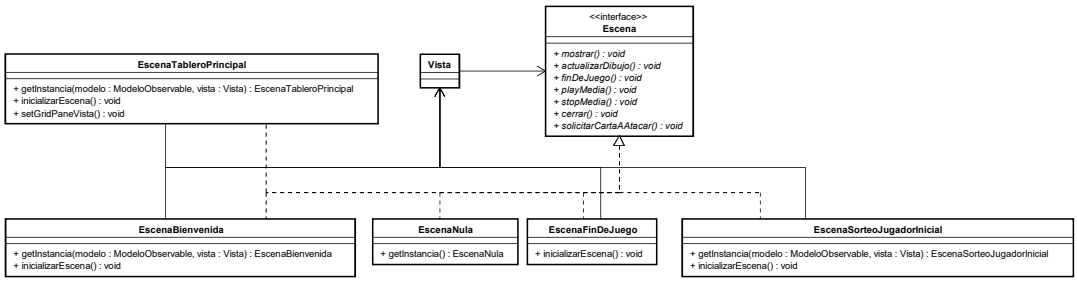

Figura 4.11: Vista.

5. Diagramas de secuencia

A continuación mostraremos y explicaremos algunas de las secuencias mas complejas presentes en el programa.

5.1. Ataque de carta sin trampas, parte MVC

En este diagrama (Figura 5.1) veremos que ocurre, del lado de la `Vista`, `Modelo` y `Controlador` cuando un usuario ataca con una de sus `CartaMonstruo` a una `CartaMonstruo` de su oponente. Vamos a suponer para este diagrama que ya fue jugada alguna carta Monstruo (Por lo que le es posible al jugador atacar) y se cumplen las condiciones necesarias para que el jugador pueda realizar un ataque.

La secuencia comienza con el jugador oprimiendo "atacar" en la carta Monstruo que se encuentra en su región durante la fase de ataque. Luego de que ocurre este evento, le llega al controlador un aviso de que el jugador quiere atacar, por lo que, luego de verificar que se cumplan todas las condiciones para que se efectúe un ataque (Si estas no se cumplen se le da un aviso al usuario), le pide al usuario que quiere atacar que elija su "Carta a atacar". Esto se realiza en la vista, haciendo que le aparezcan al jugador las opciones para su ataque, que serían las cartas enemigas a las cuales el puede atacar (Vale notar aquí que, en caso de que no haya cartas para atacar, se efectúa el ataque directamente al jugador oponente).

Una vez hecha la selección, se registra el ataque de la carta (Ya que una misma carta no puede atacar dos veces en el mismo turno) y se le delega al Modelo que efectúe el ataque. Antes de hacer esto, se ingresa a la `FaseTrampa`, en la cual se verifica si, durante este ataque, es necesario activar o no alguna `CartaTrampa`.

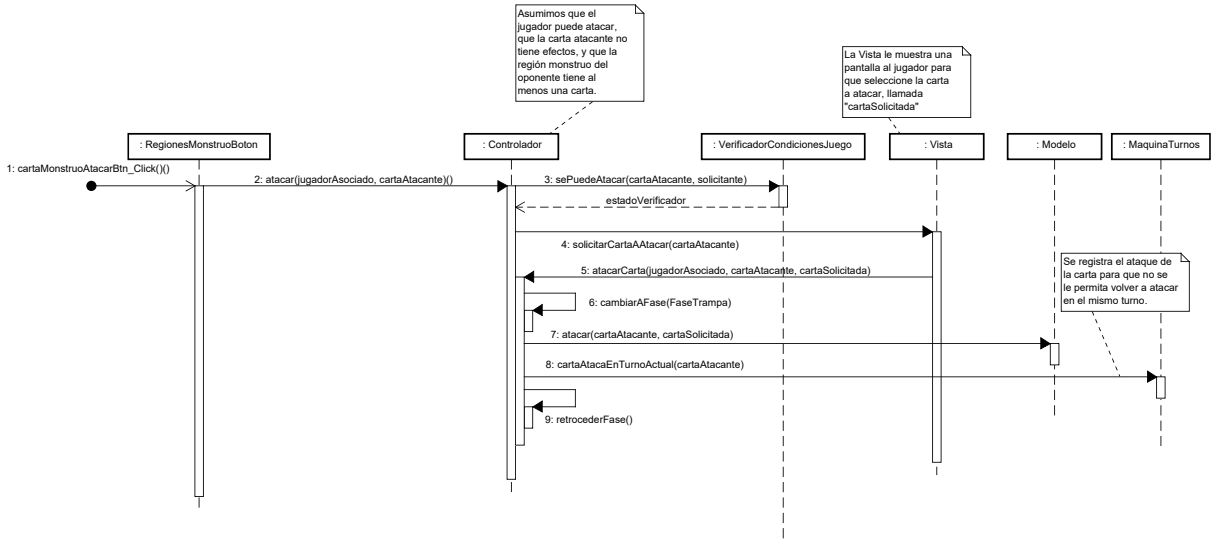

Figura 5.1: Diagrama de secuencia de un ataque de carta monstruo a otra sin cartas trampa en el campo.

5.2. Ataque de cartaMonstruo con mayor ataque a otra cartaMonstruo con menor

Este diagrama (Figura 5.2) es una continuación del anterior (Figura 5.1) solo que concentrándose en que ocurre dentro del `Modelo` cuando se efectúa un ataque.

Cuando le llega una solicitud de ataque al `Modelo`, este verifica que no haya cartas trampa en la región del oponente (Se realiza durante la `FaseTrampa` mencionada anteriormente), suponemos en este caso que no existen `CartaTrampa` en la región del oponente.

De la manera que fue implementado, una `CartaMonstruo` ataca a otra `CartaMonstruo`, esta última, "recibe" el ataque de la otra. Cuando esto ocurre, la misma `CartaMonstruo` realiza el cálculo de puntos y decide cual es el resultado de la "pelea" entre ellas cartas. En nuestro caso, la carta "atacante" tiene mas puntos de ataque que la carta "atacada", por lo que la atacada es destruida y enviada al cementerio para luego, disminuirle al jugador propietario de la carta que perdió la batalla los puntos de vida correspondientes.

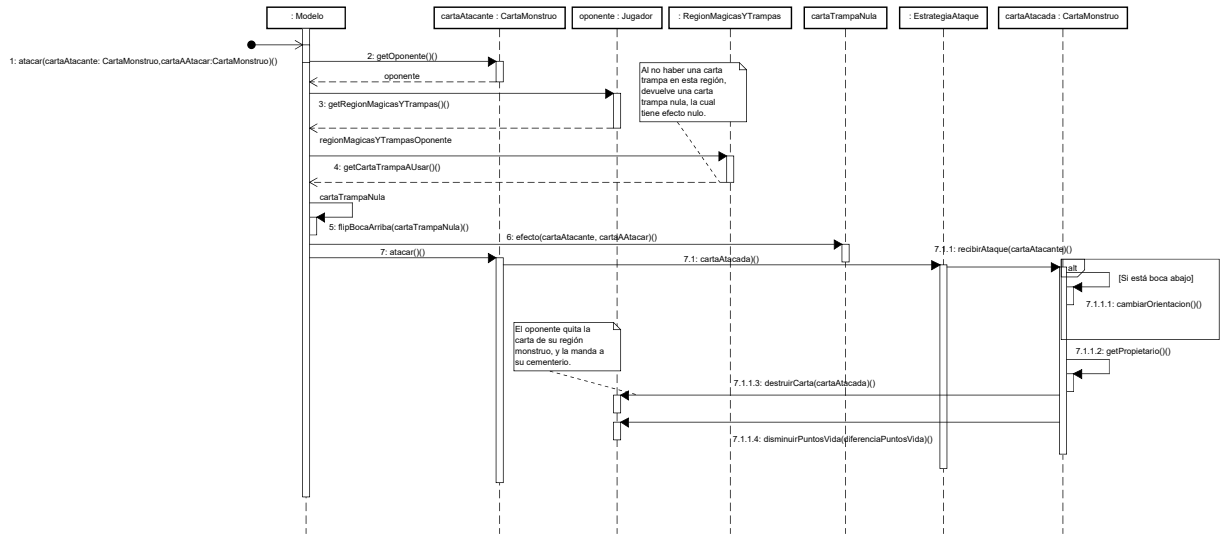

Figura 5.2: Diagrama de secuencia de ataque de una carta monstruo a otra con menor ataque.

5.3. Uso de carta Dark Hole desde la mano

En este diagrama (Figura 5.3) veremos el caso en el que la carta `DarkHole` (que es del tipo `CartaMagica`) se juega desde la mano. Especificamos que se juega desde la mano ya que existe la posibilidad de que el usuario "posicione" la carta para ser usada cuando el lo desee.

La secuencia comienza, al igual que con el caso de el ataque de la `CartaMonstruo`, desde la `Vista`. En este caso, el usuario presiona el botón correspondiente a "activar" la carta mientras esta se encuentra en su mano. Nuevamente, vamos a suponer que se cumplen todas las condiciones necesarias para que el jugador pueda activar esta carta (Que sea la fase correcta, que sea su turno, etc.). Cuando el botón se presiona, se realiza un pasaje de mensajes desde la `Vista` (que en este caso sería la `ManoVista`) por el `Controlador` hasta llegar al `Modelo`. Una vez en el modelo, se muestra la carta y se activa su efecto (Característica principal de las `CartaMagica` y `CartaTrampa`).

Como la carta tiene como atributos al `Jugador` propietario de la carta y su `Oponente`, les envía el mensaje "destruirMonstruos" a ambos, haciendo que se destruyan y manden al `Cementerio` todos los monstruos que estaban colocados en cada `RegionMonstruos`. Por último, el modelo se encarga de quitar la carta `DarkHole` de la mano del `Jugador` que la usó y enviarla al `Cementerio`.

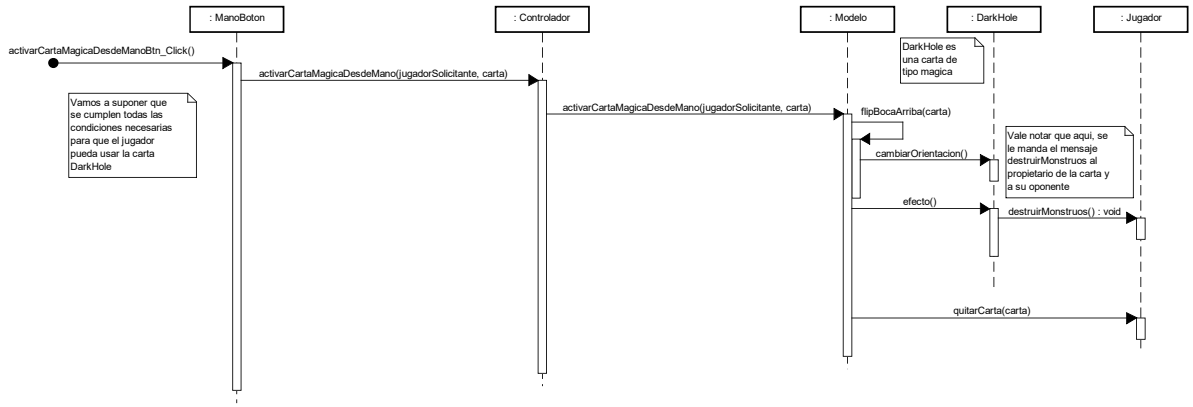

Figura 5.3: Diagrama de secuencia de activación de carta mágica Dark Hole desde la mano del jugador.

5.4. Uso de carta Monstruo come hombres

En este diagrama (Figura 5.4) podemos ver el funcionamiento de la carta `ManEaterBug` o "Monstruo come hombres".

La secuencia en sí es muy parecida al caso visto en la figura 5.2 solo que, a la hora de "recibir" el ataque, nuestra "cartaAtacada" tiene un efecto. Este efecto se activa cuando el ataque es recibido y, en el caso de `ManEaterBug`, el efecto es que la "cartaAtacante" es destruida si ataca a este monstruo mientras se encuentra boca abajo.

Lo importante a notar aquí es que, al igual que con las `CartaMagica` y `CartaTrampa`, hay algunas `CartaMonstruo` que tienen un efecto. Este efecto, al igual que con las cartas ya mencionadas, es propio de la misma y el "que hacer" del efecto esta inscripto en cada carta.

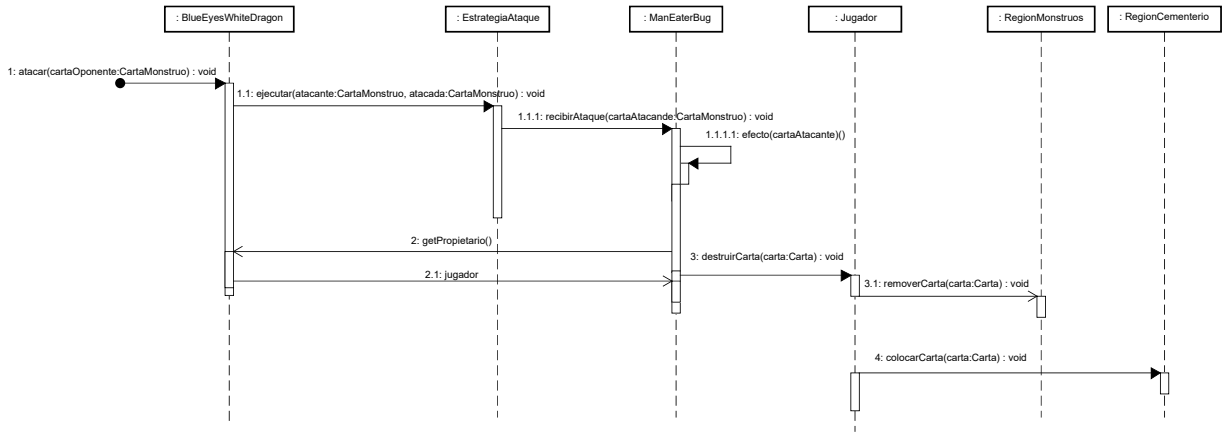

Figura 5.4: Diagrama de secuencia de carta monstruo dragón atacando a la carta Come-Hombre que se encuentra boca abajo.

5.5. Carta ingresando a region monstruos

En el diagrama (Figura 5.5) veremos que ocurre cuando ingresa una `CartaMonstruo` a la `RegionMonstruo`, específicamente las notificaciones que se realizan con los observadores. Este diagrama esta basado en una `CartaMonstruo` ingresando a `RegionMonstruo`, pero la secuencia con las cartas `CartaMagica` y `CartaTrampa` es similar.

La secuencia comienza cuando el `Jugador` ingresa una `CartaMonstruo` a la `Region`. Esta se agrega y luego, se notifica a los observadores de `RegionMonstruo` (En este caso el `Modelo`) que ingreso una carta, luego, el `Modelo` se encarga de ver de donde viene la notificación para después, notificarle específicamente a `RegionesMonstruoGrid` que ingresó una carta. Una vez recibido el mensaje, la parte de la `Escena` asociada a esta región se actualiza.

Finalmente, se le notifica a la `RegionCampo` que ingreso una nueva carta monstruo, esta región luego le pide a la `RegionMonstruo` la "ultima carta en ingresar". Una vez obtenida la carta, se le aplica el efecto de la `CartaCampo`.

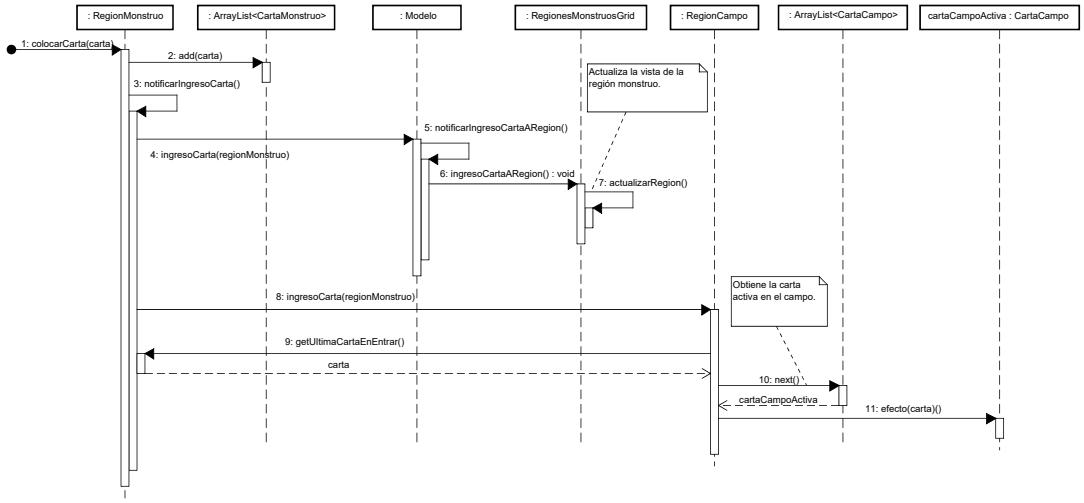

Figura 5.5: Diagrama de secuencia de una carta ingresando a la región monstruo.

5.6. Diagrama de activación de carta campo

Veremos como es el proceso en el cual se ingresa una `CartaCampo` al juego (Figura 5.6). Cuando se pide activar una `CartaCampo`, se le solicita desde el `Modelo` la `RegionCampo` del `Jugador` que solicitó activar la carta. Si esta región se encuentra vacía, se ingresa la `CartaCampo` en la misma, si ya había una `CartaCampo`, esta es reemplazada por la nueva, deshaciendo los efectos que producía la anterior.

Cuando ingresa la carta campo a la región, se le notifica a sus observadores que ocurrió este evento, por lo que de ahora en más, a cualquier `CartaMonstruo` que ingrese a `RegionMonstruos` se le aplicara el "efecto" de la `CartaCampo`. Además, cuando ingresa la carta campo, se obtienen todas las cartas monstruo en los campos del jugador y su oponente, y se le aplica el efecto de la carta campo.

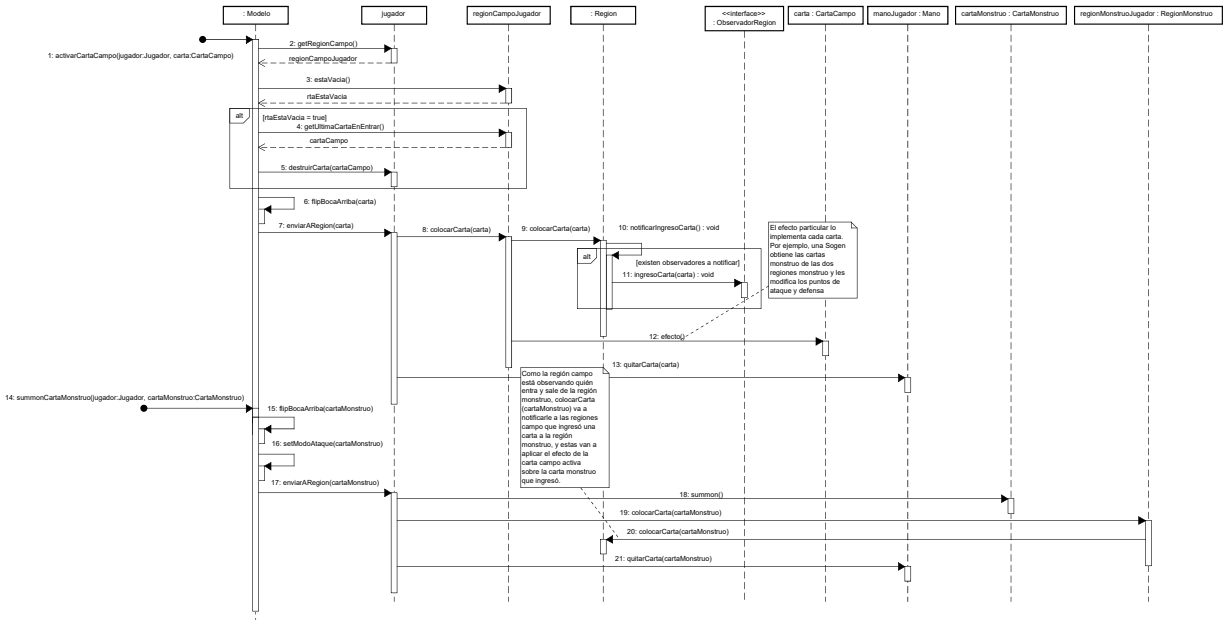

Figura 5.6: Diagrama de secuencia de la activación de una carta campo y el posterior ingreso de una carta monstruo al campo.

6. Diagramas de paquetes

En la figura 6.1 se muestra el diagrama de paquetes de la aplicación con sus dependencias internas más importantes. En general, los paquetes de excepciones suelen depender de todos los demás dentro de un paquete, ya que engloban todas las excepciones que se pueden lanzar para un dado paquete, por lo que no se especifica dicha dependencia con una flecha.

Es importante notar que en el diagrama de paquetes puede verse cómo el patrón MVC define una arquitectura donde el modelo no tiene conocimiento del controlador ni de la vista, pero que el controlador y vista dependen mutuamente entre sí.

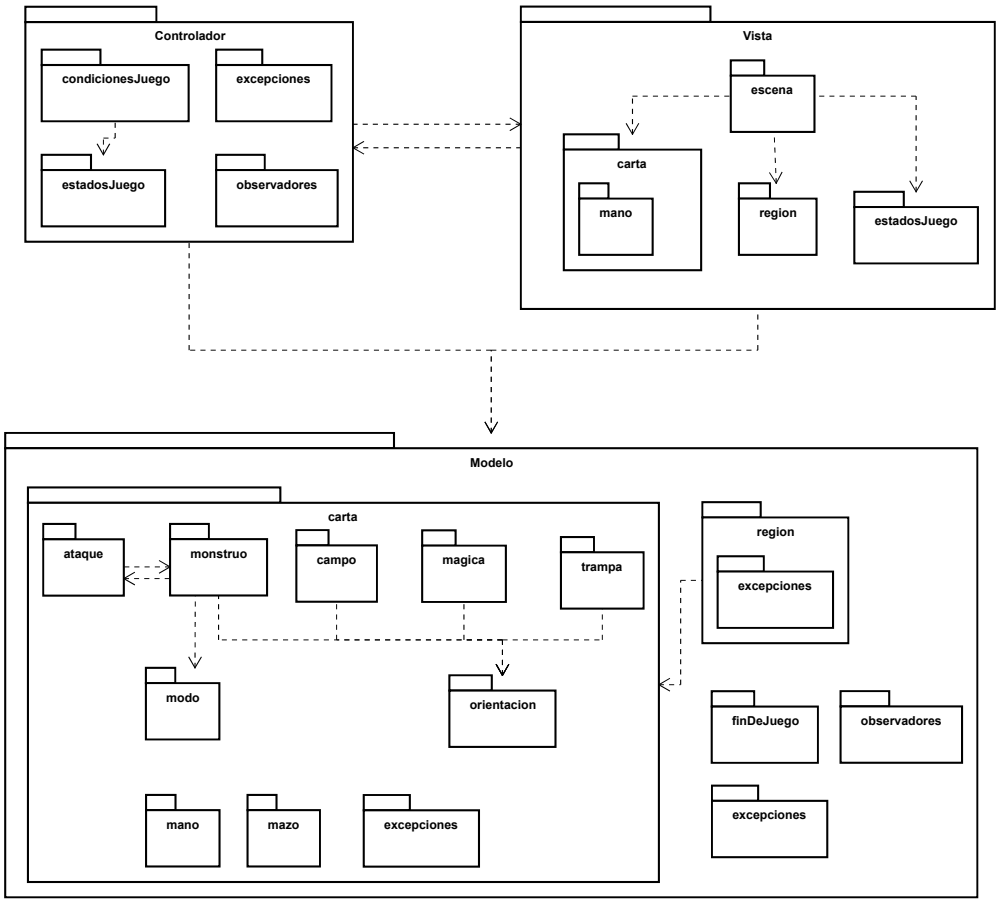

Figura 6.1: Diagrama de paquetes y dependencias.

7. Diagramas de estado

Aquí veremos como se manejan los estados de algunas de las componentes del juego. Indicando que ocurre en cada sección y como se realiza la transición a las próximas.

7.1. Diagrama de escena

Aquí (Figura 7.1), podemos ver como va cambiando la `Escena` mientras avanza el juego. Comenzamos diciendo que cada `Escena` (Ya sea de bienvenida, sorteo, tablero, etc) tiene la posibilidad de salir si se quiere, pasando a "Vista confirmación cerrar programa" de una manera u otra.

El juego comienza en la "Vista de bienvenida", en donde los usuarios podrán ingresar sus nombres. Si se presiona "jugar" en esta escena, pasamos a la "Vista de sorteo" en la cual se elige que jugador va a comenzar su turno primero.

Al finalizar esta etapa, pasamos a la "Vista tablero". Esta vista es la principal del juego y es en donde se realizan todas las interacciones entre cartas, jugadores y demás. Cuando se

dispara alguna de las `Causas Fin de Juego`, la escena pasa automáticamente a la "Vista fin de juego", en donde se muestra al ganador del juego.

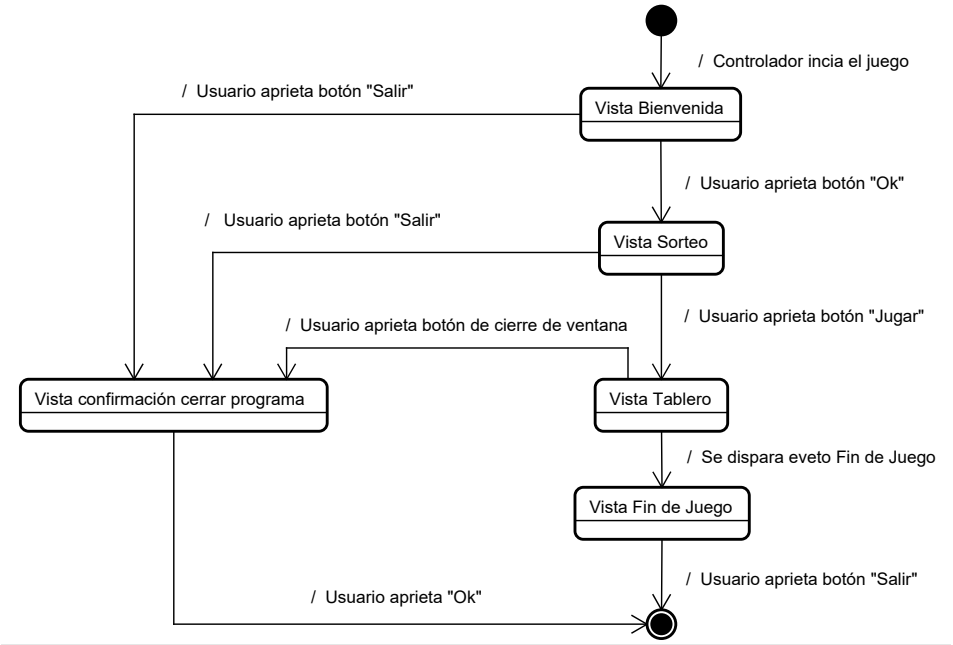

Figura 7.1: Diagrama de estado de las diferentes escenas.

7.2. Diagrama de fases

Vamos a analizar como se cicla entre las fases del juego (Figura 7.2. Notamos que esto solo ocurre una vez ingresado a "Vista tablero", previamente se eligió aleatoriamente que jugador comienza primero. También, se pueden "Saltear fases", ya que es posible terminar el turno antes de llegar a la `FaseFinal`.

Cada turno comienza con la `FaseInicial`, en la cual el `Jugador` toma automáticamente una carta. Luego de este evento, se pasa a la `FasePreparacion`. Aquí se pueden posicionar `CartaTrampa`, `CartaMagica` y `CartaCampo`, también `CartaMonstruo` pero solo una por turno. Luego de esta fase, si el jugador lo desea, puede pasar a la `FaseAtaque` en la cual, como su nombre lo indica, se le permite atacar (Como dijimos antes en el informe, no se puede atacar en el primer turno del juego). Lo interesante de esta fase es que está "anidada" con la `FaseTrampa` ya que, cada vez que se efectúa un ataque entre cartas, se pasa de la `FaseAtaque` a la `FaseTrampa` para verificar si es necesario activar algún efecto de alguna `CartaTrampa`.

Por último, pasamos a la `FaseFinal`. Aquí se les permite a los jugadores activar sus `CartaMagica` y `CartaTrampa` que hayan sido posicionadas anteriormente. Cuando esta fase se termina, se pasa automáticamente al turno del próximo jugador y el ciclo comienza nuevamente.

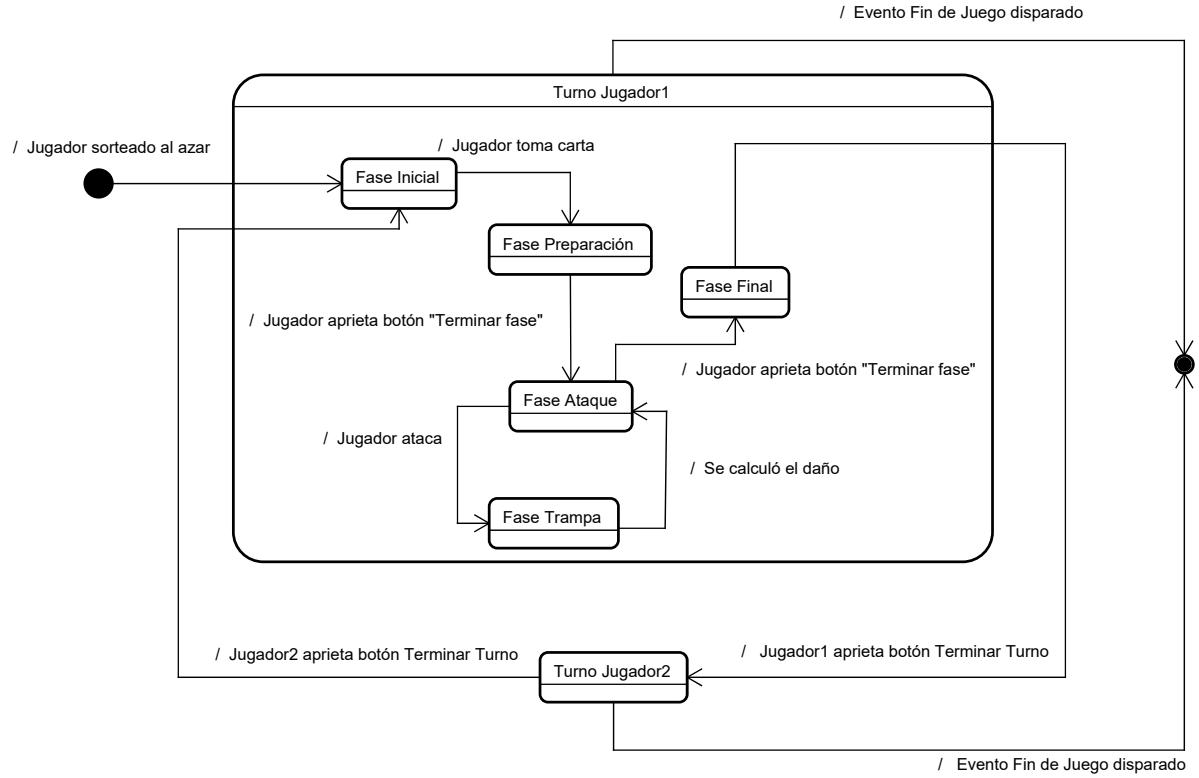

Figura 7.2: Diagrama de estado de los turnos y fases.

8. Detalles de implementación

Para el desarrollo del trabajo se utilizó la técnica de **Test Driven Development** para poder ir definiendo las entidades necesarias. Además, la organización del trabajo fue mediante **Pair Programming** para minimizar errores y tiempos de desarrollo, con reuniones tanto físicas y por videoconferencia.

8.1. Refactorizaciones

Para la realización de este trabajo práctico, se adoptó un desarrollo iterativo de mejoras ya que el código fue cambiando en reiteradas ocasiones debido a la implementación de múltiples refactorizaciones.

En un principio, se decidió que lo ideal era no utilizar una clase `Tablero` para modelar el juego y se logró la interacción entre los dos jugadores mediante la clase `Jugador`. A pesar de esto, a medida que se avanzó en la primera entrega, se tuvieron complicaciones de pasaje de mensajes y se optó por utilizar la clase `Tablero`. Luego de discutir con nuestro tutor sobre el diseño, se decidió que la utilización de un tablero como entidad arbitraria no era buena idea y que cada objeto del juego debiera comunicarse con otro para representar en mayor proporción la realidad, y no crear un árbitro/tablero que no existiese. Esto rompería con el **Principio SOLID de Responsabilidad Única** (el tablero estaría encargado de todas las acciones) y

además, con el **Principio de Inversión de la Dependencia**, ya que todos dependerían de esta entidad.

Entonces, se retomó la idea de que los dos jugadores se relacionen e interactuen entre ellos directamente y no mediante el árbitro/tablero. Se refactorizó para lograr eliminar el tablero, y se decidió que el ataque lo haga el jugador utilizando la carta y no que sea la carta la que atacaba al oponente. Esto también lo hablamos con el profesor y tomamos la decisión de que cada carta sea la que realiza el ataque y además, los efectos también sean implementados por cada carta. La implementación del efecto, en principio, la realizamos con una interfaz que obligaba a que todos tengan un efecto pero, como cada efecto era muy particular (recibía parámetros distintos) optamos por hacer que cada clase que use efecto tenga su método diferente al resto.

Durante el desarrollo de la segunda entrega, se vio la necesidad de refactorizar nuevamente y convertir a cada tipo de carta en una clase abstracta, para que los monstruos específicos hereden de ellas y estos sean los que tengan las particularidades de cada una. Dicha decisión cumple con el **Principio de Sustitución de Liskov**, donde todas las clases hijas (Monstruos) son cartas monstruo y las cartas monstruo, son cartas. Esto aplica para toda la relación de cartas, y cumple el principio ya que todas las clases hijas se pueden ubicar en lugar de la clase padre.

También se refactorizó el cálculo de daño y ya no son las cartas monstruos las que restaban puntos al jugador, sino que el jugador se quita sus propios puntos de vida, cumpliendo así con el **principio SOLID abierto/cerrado**, ya que la clase debe estar cerrada al exterior para su modificación, pero abierta para su extensión.

Durante la preparación de la entrega final se volvió a aplicar una refactorización que consistía en que el jugador (que hasta entonces, era el encargado de atacar y de llevar a cabo las tareas) ya no sería el encargado de implementar las estrategias de las cartas, sino que ahora ellas mismas son las que atacan y activan sus efectos. Esta refactorización, también cumple con el **Principio de Responsabilidad Única**, debido a que el jugador le hace pedidos a las cartas pero no le interesa como las cumplen.

Una vez finalizado el modelo, se comenzó a implementar la interfaz gráfica y la idea con la que se comenzó fue conectarla directamente con el modelo pero nos dimos cuenta de que el usuario iba a interactuar con mas de lo que debía. Esta idea rompía con el **Principio de Segregación de la Interfaz** ya que el cliente solo debiera conocer lo que él solamente necesita, y luego de reunirnos con nuestro tutor, se decidió utilizar el patron de arquitectura MVC para separar correctamente las responsabilidades, minimizar dependencias, y aprovecha la segregación de interfaces.

9. Excepciones

En esta sección hablaremos de las diferentes excepciones que fueron usadas en el programa, listándolas y explicando cuando ocurren cada una de ellas ya que varías son utilizadas para diferentes casos a través del programa. Cada uno de estos errores se manifiesta con una "alerta" en la vista, avisándole al usuario que ocurrió. De la manera que se manejó, dependiendo el caso en donde apareció la excepción, la alerta mostrada es diferente. Por ejemplo, lo que ocurre con `NoSePuedeAtacarError` a continuación.

9.1. Controlador

- NoSePuedeAtacarError

  Este error ocurre cuando el jugador intenta atacar cuando no debe. Existen dos casos en donde este error puede aparecer, el primero es cuando se intenta atacar en una fase que no es la `FaseAtaque` y el otro, cuando se intenta atacar con una `CartaMonstruo` que ya atacó en el turno. Estos casos se diferencian con la alerta enviada. Si se dio el primer caso de la excepción, se muestra una alerta, si se dio el segundo, otra diferente.

- NoSePuedeCambiarOrientacionError

  El error ocurre principalmente cuando se intenta cambiar la orientación de una carta que ya fue cambiada de orientación en el mismo turno, acción que es invalida con respectos a los supuestos presentados anteriormente.

- NoSePuedeEnviarARegionCampoError

  Ocurre cuando se intenta jugar una `CartaCampo` en una fase que no es la `FasePreparacion`.

- NoSePuedeEnviarCartaMonstruoARegionError

  Ocurre cuando se intenta jugar una `CartaMonstruo` en una fase que no es la `FasePreparacion`.

- NoSePuedeEnviarMyTARegionError

  Ocurre cuando se intenta jugar una `CartaTrampa` o una `CartaMagica` en una fase que no es la `FasePreparacion`.

- NoSePuedeUsarMyTError

  Este error se da cuando se intenta usar una `CartaMagica` o `CartaTrampa` que fue posicionada en una fase que no es la `FaseFinal`

- SeTerminaronLasFases

  El error se da cuando se llega a la `FaseFinal` y el `Jugador` presiona “Avanzar fase”, en este caso no se le notifica el error al usuario con una alerta, sino que pasa automáticamente al turno del próximo `Jugador`.

9.2. Modelo

- SacrificiosInsuficientesError

  Como su nombre lo indica, este error se da cuando el `Jugador` intenta “invocar” una `CartaMonstruo` que requiere `Sacrificio` pero no hay suficientes cartas en la `RegionMonstruos` para sacrificar.

10. Conclusiones

Se desarrolló una aplicación que implementa de manera simplificada el juego de cartas Yu-Gi-Oh! utlizando el lenguaje de programación Java y la plataforma JavaFX para la interfaz gráfica.

Se estudiaron diferentes patrones de diseño que resultaron de utilidad para modelar varias partes del programa, y en particular, el patrón Observer ya que permitió desarrollar adecuadamente los diferentes eventos que había que procesar y también minimizar acoplamientos entre clases.

Por otro lado, el uso del patrón de arquitectura MVC nos permitió desarrollar independientemente el controlador y la vista a pesar de que se realizaban en paralelo varias refactorizaciones del modelo.

Por último se logró aprender cómo la programación orientada a objetos permite identificar entidades independientes minimizando el acople entre ellas y facilitando el desarrollo de aplicaciones de complejidad media a elevada.

A. Referencias

[1] Fontela, Carlos - *Programación Orientada a Objetos con Smalltalk, Java y UML.* - 3ra edición - Versión Beta 0.6.

[2] Fowler, M. - What's a model for?

[3] Yu-Gi-Oh! TRADING CARD GAME rulebook