FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Al-Go-Ho!

[7507/9502] Algoritmos y Programación III
Curso 1
Primer cuatrimestre de 2018

Grupo: T8		
Nombre	Padrón	Mail
Goijman, Lautaro Enzo	101185	lautiman@gmail.com
Petrignani Castro, Ignacio	100863	ipetrignani@fi.uba.ar
Peña, Esteban	100102	tatas323@gmail.com
Curzel, Martin	100049	martin.curzel@gmail.com

Índice

1. Introducción

El presente informe reúne la documentación de la primer entrega 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!, este se desarrolla en el lenguaje Java, utilizando los conceptos del paradigma de la orientación a objetos vistos en el curso.

2. Supuestos

Para realizar de forma correcta la resolución del trabajo practico, implementamos toda la teoría dada en clase para obtener un funcionamiento adecuado, es decir utilizando el paradigma de la programación orientada a objetos. Un Primer supuesto es implementar un juego de mesa interactivo en el cual participa mas que un jugador. Es decir, que debemos encontrar la forma de pasar los turnos y no mezclar la información que tiene cada jugador. Otro supuesto fue encontrar la forma correcta de activar los efectos de las cartas de utilidad como las de trampa. También surgió el problema de como realizar una interfaz amigable a la hora de tener que jugar. Finalmente otro supuesto fue la carta de fusión, que habilita al jugador hacer una fusión de varias cartas a una sola que no existe en el mazo.

3. Modelo de dominio

El desarrollo del juego consiste en el trabajo grupal, por lo tanto diseñamos un sistema donde cada integrante del equipo puede entender la implementación del otro. Decidimos hacer la mayo cantidad de abstracción y mantener el encapsulamiento entre los objetos. El objetos principal del juego es el jugador, ya que el mismo se encarga de delegar al resto de los objetos los mensajes para realizar su turno. También fue importante detectar las diferencias entre las distintas cartas, por lo tanto desarrollamos una interfaz carta que tiene el la misma el funcionamiento fundamental de sus hijas. Otro objeto muy importante es el lado, en este se guardan las cartas en mesa. Cada jugador tiene un lado de la mesa donde el mismo hace las distintas jugadas con su mazo y mano. Una carta puede afectar los dos lados de la mesa, por lo tanto un jugador tiene el acceso a ambos lados de la mesa.

Al tener que implementar un juego de mesa supimos que los objetos Carta, Mazo y Jugador tienen que estar en la misma. Creando dichos objetos y entendiendo a como modelar el juego de Yu-Gi-Ho!,fuimos agregando las distintas cartas y que pueden ser utilizadas por el usuario. Ahí fue cuando creamos las cartas de utilidad y las cartas monstruos. En nuestro trabajo se puede

ver claramente que se divide en juego en dos lados (uno por cada jugador), el lado tiene todos los objetos necesarios para poder realizar el juego. El lado representa la forma en la que el jugador ve sus cartas en una mesa en la vida real. Esta mesa consiste de las siguientes entidades:

Mazo El mazo del jugador.

Cartas Monstruo Cartas monstruo jugadas(en mesa) por el jugador.

Cartas Trampa O Mágicas Cartas de utilidad jugadas(en mesa) por el jugador.

Mazo De Fusiones Cartas de cartas para fusionar.

Carta De Campo Carta de campo jugada(en mesa) por el jugador.

Fusion Estado de fusión.

La parte mas complicada fue mantener la mayor abstracción en nuestra implementación, por lo tanto agregamos varias clases que ayudan a delegar responsabilidad del jugador. El jugador tiene estas entidades:

Puntos De Vida Puntos de vida del jugador(8000 iniciales y 0 cuando pierde).

Mano Cartas en mano del jugador, es decir con opción a jugar en su lado.

Lado Enemigo Tiene la información del lado enemigo al cual ataca.

Enemigo Puede atacar directamente al enemigo contrario.

Lado Lado propio del jugador.

Fase Las distintas faces del turno.

4. Diagramas de clase

En el siguiente diagrama de clase(Figura 1) muestra una breve y concisa relación entre las clases utilizadas para realizar el juego. Se puede observar que la clase principal es la del Jugador. También la implementación de la interfaz carta.

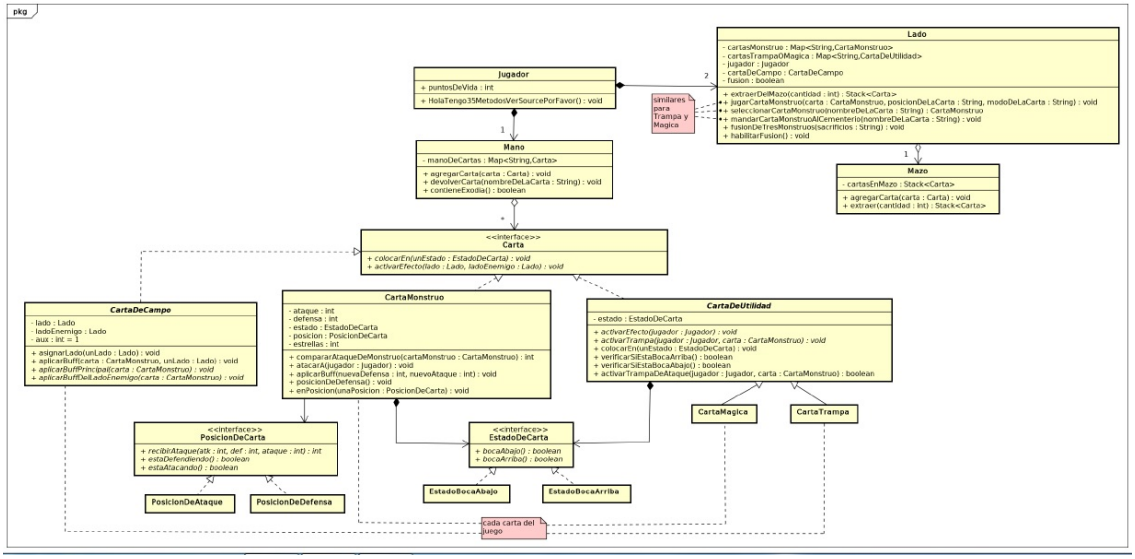

Figura 1: Diagrama de Al-Go-Oh!.

5. Diagramas de secuencia

En el siguiente diagrama de secuencia(Figura 2) muestra como se comportan los objetos cunado el jugador ataca a su enemigo. Se puede ver como si o si el monstruo tiene que cambiar el estado a boca arriba y en posición de ataque para realizar el ataque. Luego rival recibe el daño directo bajando sus puntos de vida.

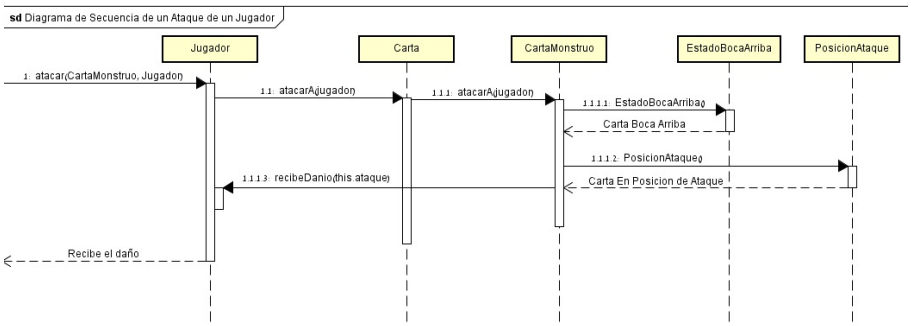

Figura 2: Ataque a Enemigo.

En el siguiente diagrama de secuencia(Figura 3) muestra como se comportan los objetos cunado el jugador ataca un monstruo enemigo. Se puede ver como el monstruo cambia su estado a boca arriba y posición de ataque para realizar el ataque. Luego el monstruo enemigo recibe el ataque y en caso de tener vida negativa el resto del ataque se aplica directamente al rival enemigo.

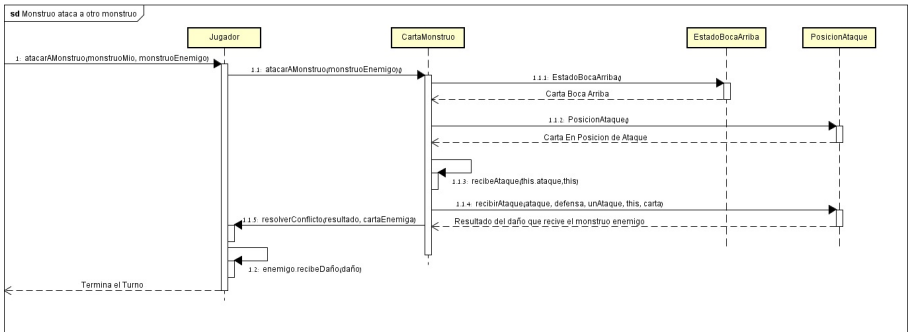

Figura 3: Ataque A Monstruo.

En el siguiente diagrama de secuencia(Figura 4) muestra como se activa la trampa mediante el ataque del rival. Antes de realizar el ataque del monstruo se activa(si existe) la trampa del rival y se realiza el efecto de la misma. Luego se sigue el ataque con el efecto en ya activo en el juego.

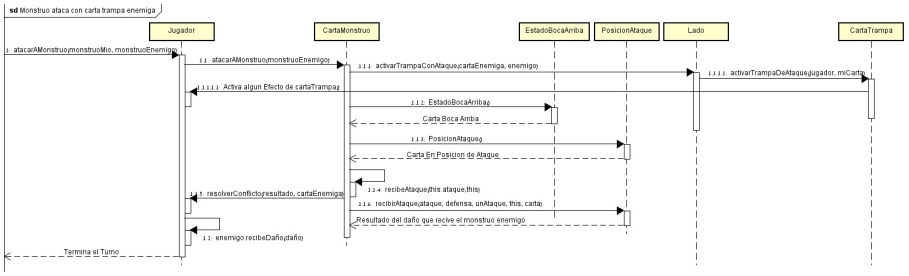

Figura 4: Activar Trampa.

6. Diagramas de paquetes

En el siguiente diagrama de paquetes(Figura 5) muestran los dos paquetes principales del juego. El modelo tiene todas la entidades que se encargar del funcionamiento junto a las cartas y excepciones. En la vista se encuentra toda la parte gráfica del juego. La vista contiene las pantallas y los distintos escenarios del juego. Cada una de estas pantallas utiliza los distintos botones. Los bonotes principales son las cartas y los mazos de cada jugador.

La vista se encarga de utilizar el modelo y darle "vida.ª las distintas entidades mientras que el modelo se encarga de la lógica de las mismas.

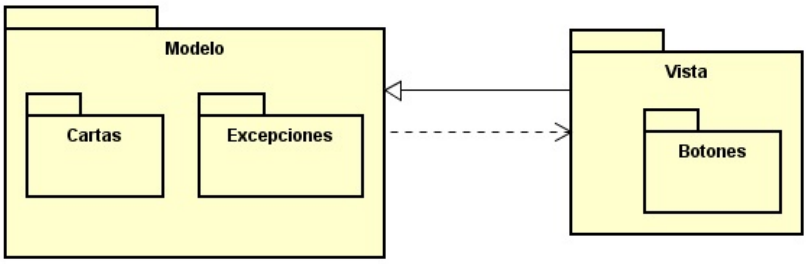

Figura 5: Diagrama de paquetes de Al-Go-Ho!.

7. Diagramas de estado

En el siguiente diagrama de Estado(Figura 6) muestra las distintas etapas del juego. Al comenzar el juego empieza el turno del primer jugador e automáticamente pasa a la primer fase del turno. En la fase inicial se coloca el lado del jugador y luego se pasa a la fase principal donde el jugador tiene las distintas opciones para poder jugar. Luego el usuario puede seguir a la fase de batalla para atacar a su rival. Finalmente se pasa a la fase final donde se termina el turno y se le pasa al siguiente jugador un turno nuevo. En todo momento el jugador puede terminar su turno. El juego termina cuando en algún turno algún jugador ya no tenga puntos de vida, ahí es cuando se anuncia el ganador y termina el juego.

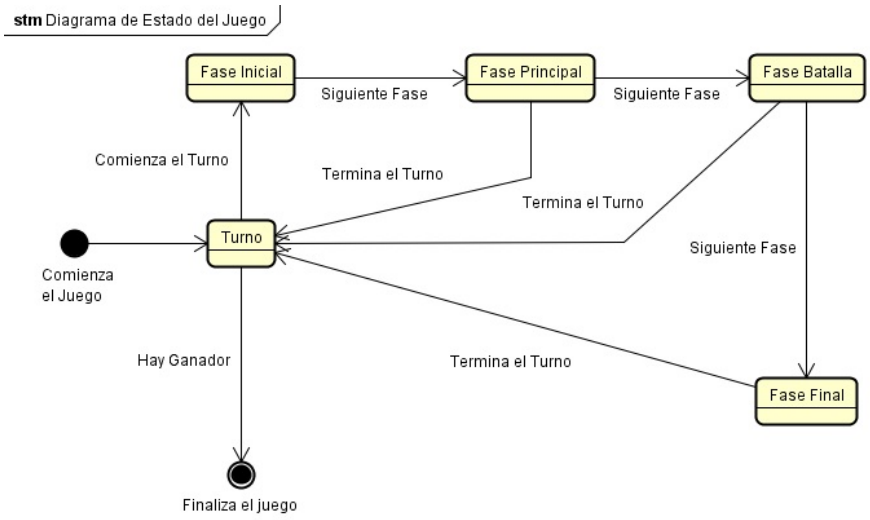

Figura 6: Diagrama de estado del Turno.

8. Detalles de implementación

En nuestro trabajo utilizamos varios patrones de diseño ya mejoraban en varios aspectos la calidad del código, legibilidad y funcionalidad. También implementamos los distintos objetos utilizando TDD y luego las pruebas de integración necesarios para comprobar el correcto funcionamiento del juego. Desarrollamos interfaces para los objetos que tienen funcionamientos parecidos tal como las cartas o faces del turno.Para evitar el uso de algunos condicionales y mas importante evitar el código repetido, elegimos usar varios casos de polimorfismo.

Estos son los detalles mas importantes del trabajo practico:

Patron State Este fue implementado para los estados de las cartas en juego. Como las cartas tienen un comportamiento dinámico a lo largo del juego, decidimos agregar estados de defensa y ataque o boca arriba y boca abajo. Así evitamos el uso de algunos condicionales. También se utilizo en el sistema de fases de cada turno. Usamos la clase fase como interfaz que luego es implementada por: Fase inicial, principal, batalla y final.

Double Dispatch Este fue implementado a la hora de que un monstruo ataca a otro. El atacante se recibe por parámetro el monstruo enemigo y luego ejecuta el segundo Dispatch en el cual la carta enemiga recibe el ataque de la carta. Así mismo evitando romper el encapsulamiento.

```
public int atacarAMonstruo(CartaMonstruo cartaMonstruo)
throws CartaMuertaNoPuedeAtacarException,
MonstruoNoPuedeAtacarDosVecesEnUnTurnoException {
    estado = new EstadoBocaArriba();
    posicion = new PosicionAtaque();
    if (yaAtacoEsteTurno)
        throw new MonstruoNoPuedeAtacarDosVecesEnUnTurnoException();
    yaAtacoEsteTurno = true;
    vidaDeCarta.atacar();
    return cartaMonstruo.recibeAtaque(this.ataque,this);
}

private int recibeAtaque(int unAtaque, CartaMonstruo carta){
    vidaDeCarta.atacar();
    estado = new EstadoBocaArriba();
    activarEfectoAlRecibirAtaque(carta);
    int resultado = posicion.recibirAtaque(ataque, defensa,
    unAtaque, this, carta);
    return resultado;

}
```

TDD Cada objeto principal de nuestro juego tiene sus propias pruebas unitarias, así nos aseguramos de un correcto funcionamiento, gran cobertura del sistema y también ayuda al compañero a entender el código.

Uso de interfaces Para evitar repetir código y ser mas organizado creamos varias interfaces. También se utilizaron para los distintos patrones de diseño.

Package java.util Usamos varias colecciones para mejorar el código de nuestro juego y evitamos desarrollar algunos objetos triviales sin comportamiento. Por ejemplo el mazo: Un stack(cola) con las distintas cartas. Map(Hash) para guardar las cartas en juego.

Polimorfismo En varios lugares surgieron los usos de condicionales. Una excelente forma de evitarlos y preguntar o violar el encampsulamiento es el uso de la POO y los polimorfismos.

Finalmente una vez que se comprobó el correcto funcionamiento del juego, se paso al desarrollo de la interfaz gráfica. Se usaron las librerías de JavaFX para la vista del juego. El juego consta de una pantalla principal y el juego. Para el juego se utilizaron varios objetos para simular una mesa de juegos

Esta(Figura 7) es la pantalla inicial del juego.

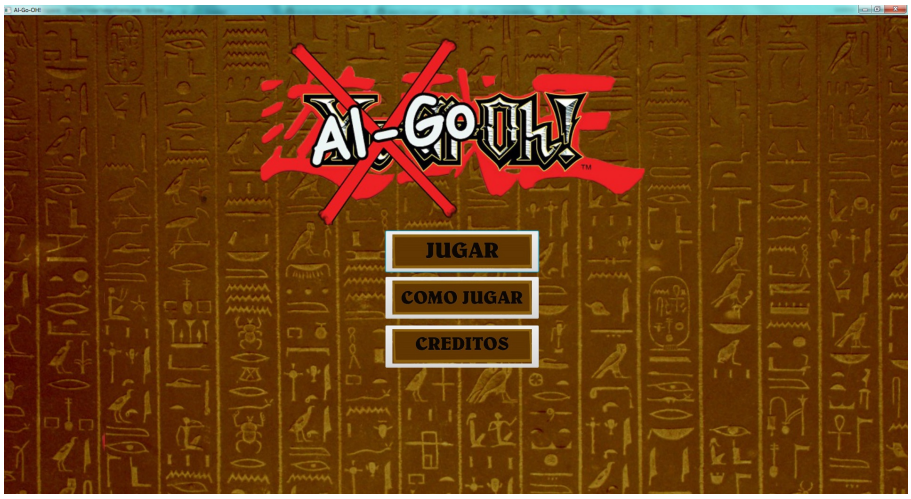

Figura 7: Pantalla Principal.

La pantalla del juego consta con un BorderPane, este nos permite dividir y manejar fácilmente los espacios en pantalla. A la izquierda se muestra la información de los jugadores y las opciones de pasar de fase como terminar el turno. En el centro del Pane se muestra el tablero completo con ambos lados. Se pueden ver(Figura 8 y 9) los distintos componentes del lado de forma clara, para que el usuario sepa en todo momento que cartas están en juego y en que posición. La mano del jugador aparece en la parte inferior de la pantalla, para la misma se utilizo un Hbox de JavaFX ya que nos permite agregar cartas como botones de forma fácil e infinita(contiene ScrollBars). Luego nos pareció necesario agregar en el costado derecho la información que contiene cada carta. Esto facilita la lectura de los datos y mejora visualmente al juego.

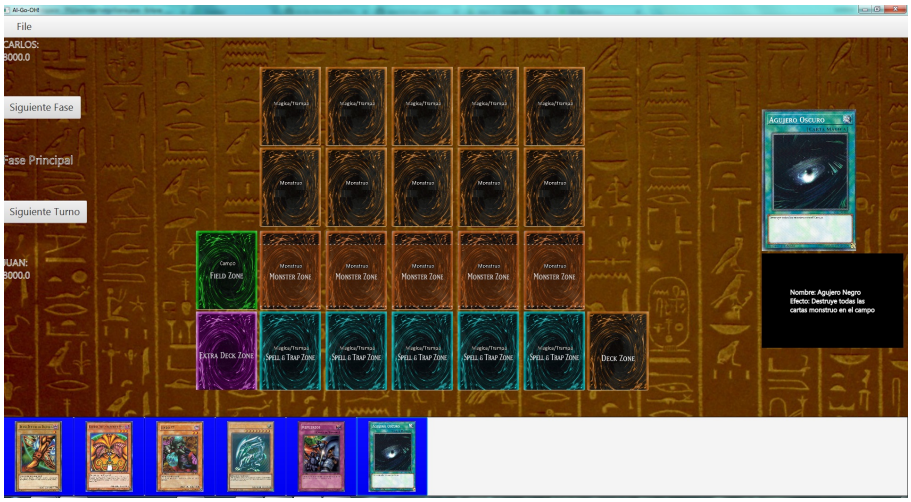

Figura 8: Pantalla de juego.

En esta imagen(figura 9) se observa en forma aún más detallada una posible instancia del juego. Además aparecen las opciones de jugar la carta en mano.

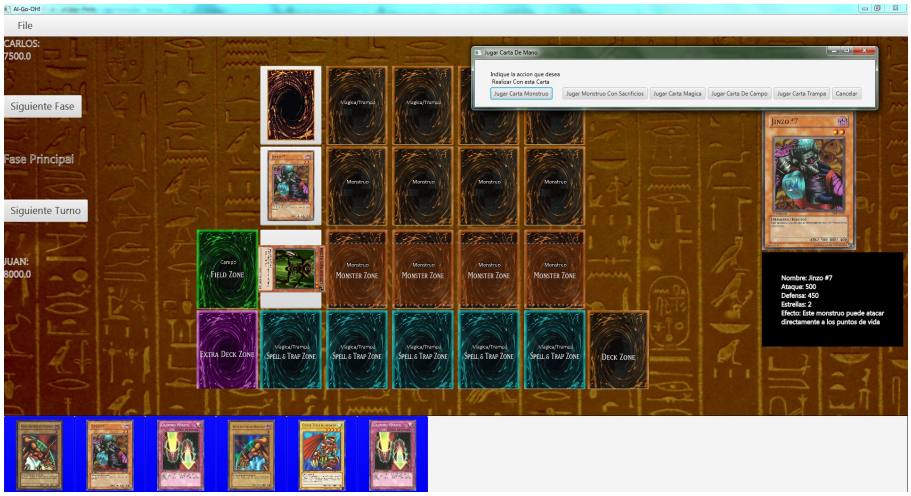

Figura 9: Pantalla de mesa con cartas.

En esta imagen(figura 10) se observa como el jugador fusiona sus tres dragones junto utilizando la carta de Polimerizacion.

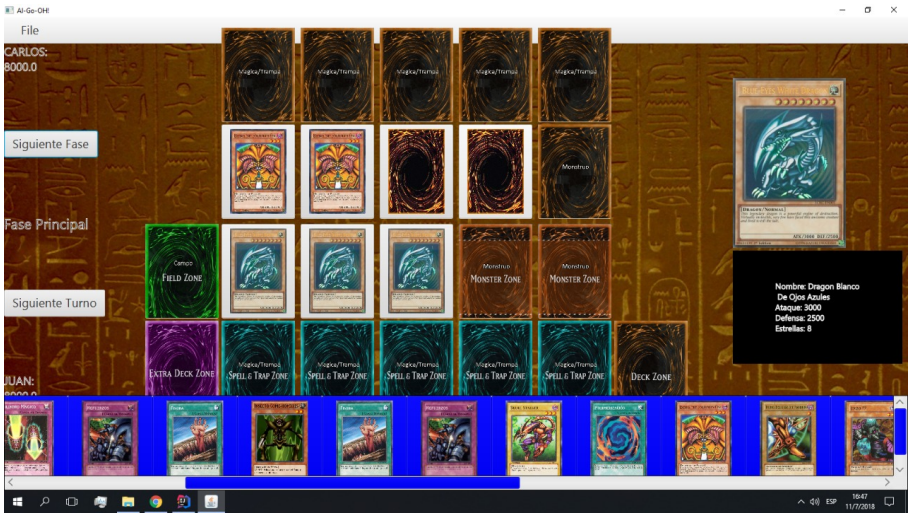

Figura 10: Tres dragones de ojos azules.

En esta imagen(figura 11) se observa el resultado de la fusión.

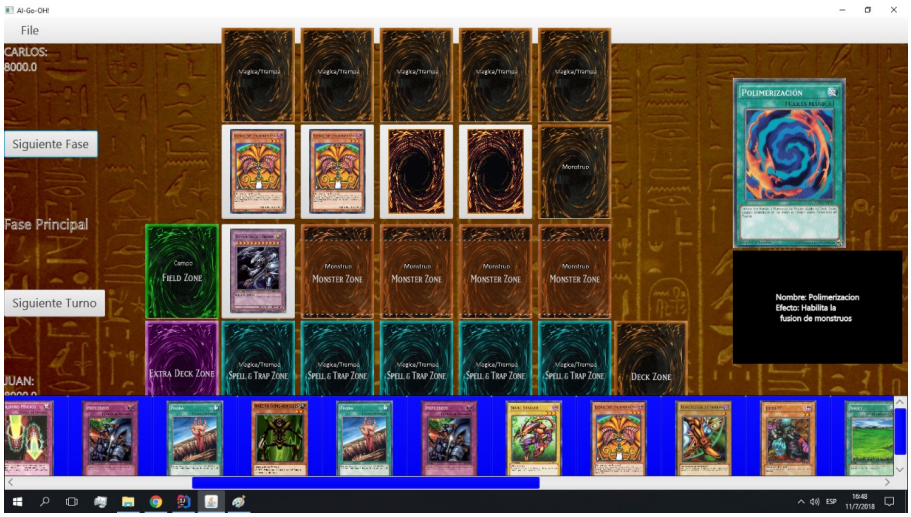

Figura 11: Dragon definitivo de tres cabezas.

9. Excepciones

Es importante lanzar y poder captar errores para un correcto funcionamiento y para evitar bugs mal entendidos a la hora de jugar. Varias de estas excepciones son lanzadas al usuario cuando esta jugando, ya que pueden generase infinitas situaciones y casos bordes. Estas son lanzadas como mensajes en pantalla de nuestra interfaz gráfica(JavaFX).

AccionInvalidaEnFaseException: Esta es lanzada cuando el usuario quiere hacer alguna acción que no corresponde a la fase actual como por ejemplo atacar en la fase final del turno.

CantidadDeSacrificiosIncorrectaException: Esta le notifica al usuario que la cantidad de sacrificios no es suficiente para convocar la carta elegida.

CartaMuertaNoPuedeAtacarException: No se puede atacar con un monstruo muerto.

ElMazoEstaVacioException: Esta se dispara cuando el mazo esta vació ya que no quedan mas cartas para sacar. En el caso del jugo el usuario es notificado y en su siguiente turno perderá ya que no tiene mas cartas en su mazo.

HayUnMonstruoEnElCaminoException: Esta evita que el jugador pueda atacar a su rival(Ataque directo) habiendo monstruos en el lado enemigo.

LadoNoContieneCartaException: En el caso de que el lado de algún jugador este vació se lanza esta excepción.

ManoNoContieneCartaException: Disparado en el caso de que el jugador no tenga cartas en su mano para jugar.

MonstruoNoPuedeAtacarDosVecesEnUnTurnoException: Evita que se pueda atacar mas de una vez en el turno con el mismo monstruo.

NoAdmiteFusionException: Esta excepción se dispara en el caso de que el jugador quiera fusionar sus cartas sin haber usado previamente la carta de Polimerizacion.

NoEsTuTurnoException: Se muestra en el caso de que un jugador quiera jugar fuera de su turno.

NoHayEspacioEnLadoException: Cuando no hay mas espacio en el lado es necesario avisar que no puede agregar más cartas en la mesa de juego.

NoHayMasFasesException: Evita que el jugador entre en alguna fase equivocada.