FIUBA - 75.07

Algoritmos y programación III

Trabajo práctico 2: Al-Go-Oh!

1er cuatrimestre, 2018

(trabajo grupal de 4 integrantes)

Alumnos:

Nombre	Padrón	Mail
Nicolás Rivas	95754	nicolasrivas1991@hotmail.com
Nicolás Sánchez	98960	sanchez_nico@hotmail.com
Gastón Taborda	96491	gastonlucastaborda@gmail.com
Tomás Santa María	92797	tomasisantamaria@gmail.com

Fecha de entrega final: 28/6/2018

Tutor: Federico Jure

Comentarios:

Supuestos

- Durante una fase de trampa, el jugador no puede elegir qué carta de trampa activar. Esto es determinado según el orden en el que fueron colocadas en el tablero.
- Algo similar ocurre con los criterios de invocación. Un monstruo con una invocación que requiere sacrificios sacrificará a la cantidad requerida de monstruos según su valor de ataque. Existe un caso especial, el del Dragón definitivo de ojos azules, que rompe con esta regla (debe sacrificar monstruos específicos para poder ser invocado).
- Suponemos que el juego está finalizado cuando a un jugador se le acabaron todos los puntos de vida. Esa es la condición primaria de victoria del juego.
- Asumimos la decisión de representar un ataque de una carta a otra (o al jugador) como dos fases distintas: una elemento que ataca y el otro que recibe el ataque.
- Las cartas de tipo mágicas jugadas desde la mano del jugador, activan su efecto de forma instantánea. Si son colocadas en la zona de mágicas/trampas la única posición válida es boca abajo.
- Un jugador solo puede colocar en su lado del tablero hasta un máximo de 5 monstruos.
- Las efectos realizados por una carta de campo son permanentes e irreversibles, aún cuando es enviada al cementerio como consecuencia de que el jugador jugó una nueva carta de campo. El mismo efecto ocurre con las cartas de trampa: sus efectos son permanentes una vez activados.

Modelo de dominio

- **Juego**: La clase principal del sistema y su punto de entrada. Las responsabilidades primarias del Juego incluyen: inicialización de variables, resolución de fases según la progresión del juego, y determinar si el mismo debe terminar y hay un ganador.
- **Jugador**: Representa a un participante del juego. Es quien tiene el Tablero sobre el cual se irán jugando las distintas Cartas, y quien tiene en su poder tanto la Mano de cartas disponibles para ser jugadas como el Mazo de cartas del cual se toman nuevas.
- **Carta:** Interfaz que representa un contrato común a todas las cartas que pertenecen al juego. Puede ser colocada en el tablero.
    - **CartaMagica, CartaTrampa, CartaCampo**: Distintas interfaces que extienden a Carta y representan los distintos tipos de cartas que pueden existir en el juego. Tienen efectos que pueden ser activados.
    - **CartaMonstruo**: Clase abstracta que representa a los monstruos del juego. Pueden atacar o ser atacados, colocarse en el tablero de distintas formas (Ataque/Defensa, Boca Arriba/Boca Abajo), y además pueden o no tener efectos de volteo (se activan cuando un

monstruo pasa de estar boca abajo a boca arriba).

- **Zonas**: Distintos lugares del tablero donde pueden ir los distintos tipos de cartas ya mencionados.
    - **ZonaDeMonstruos, ZonaDeCampo, ZonaDeTrampasYMagicas**
- **Tablero**: Simboliza al lugar donde se colocan las cartas jugadas por el Jugador. Se encarga de ubicar los distintos tipos de cartas en las distintas zonas habilitadas, y las mueve de una zona a la otra según la resolución de los distintos eventos.
- **EstrategiaLadoCarta**: Interfaz que en conjunto con EstrategiaMonstruo resuelve tanto el atacar como el recibir un ataque, según la combinatoria posible de posiciones/modos que puede tener una carta monstruo.
- **EstrategiaMonstruo**: Interfaz utilizada para resolver los ataques de las cartas monstruo según el modo en el cual hayan sido colocados en el tablero.
- **EstrategiaInvocacion**: Contrato para las cartas de tipo monstruo, cuya invocación puede alterar el estado del Tablero.
- **EstadoJugador:** Interfaz que representa el resultado (y sus consecuencias) de un ataque de un monstruo. Según la resolución del "duelo", las distintas implementaciones deciden de qué manera alterar el estado de cada jugador.
- **EstadoDeCampo, EstadoDeTrampas:** Representaciones en forma de interfaz de los distintos estados en los que se pueden encontrar las Zonas (vacíos u ocupados). Las distintas implementaciones tienen la responsabilidad de colocar nuevas cartas y evaluar si una carta puede ser invocada en la zona (según por ejemplo, la regla de máximo 5 cartas de trampa en el tablero al mismo tiempo).
- **AtaqueMonstruo, AtaqueDirecto:** Los distintos tipos de ataque que una carte de tipo monstruo puede realizar.
- **Mano:** Es el conjunto de cartas que el jugador tiene a su disponibilidad.
- **Mazo:** Colección de cartas con la cual cada jugador comienza el juego. Cada turno, el jugador toma una carta del mazo para colocarla en su mano.

Diagramas de clases

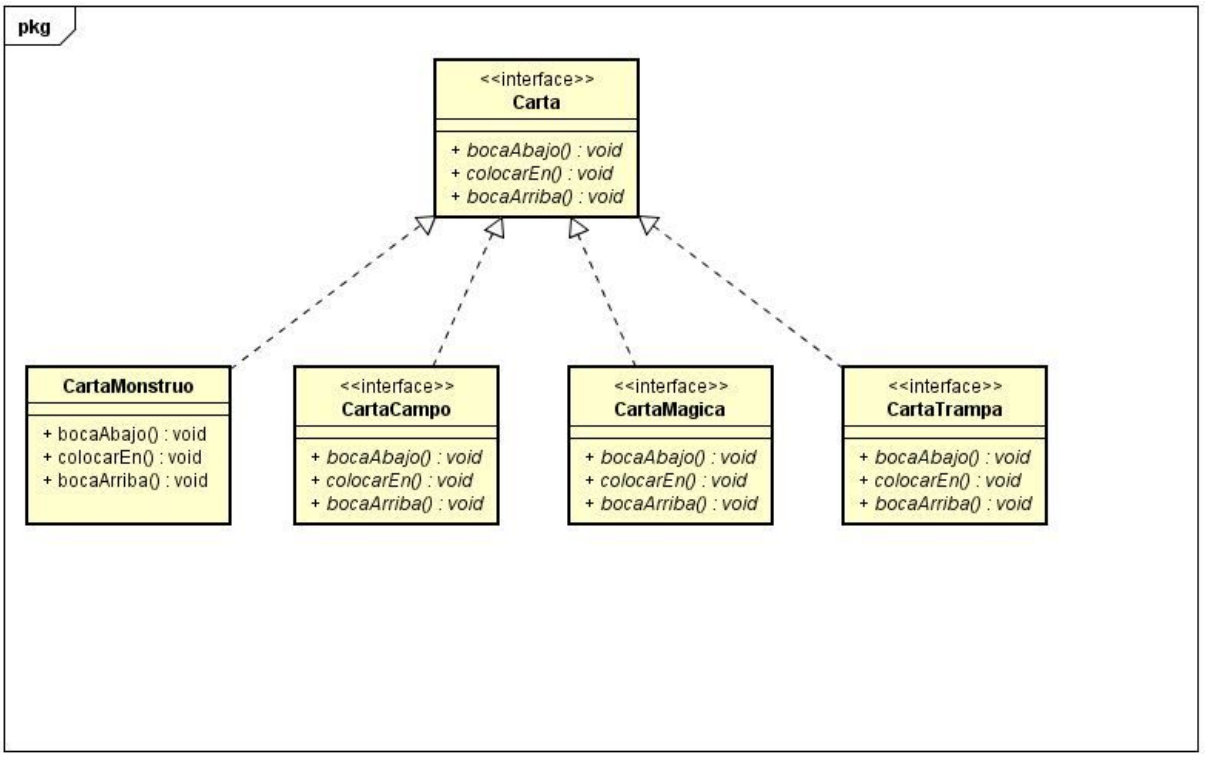


Diagrama de la clase carta, mostrando las clases que la implementan.

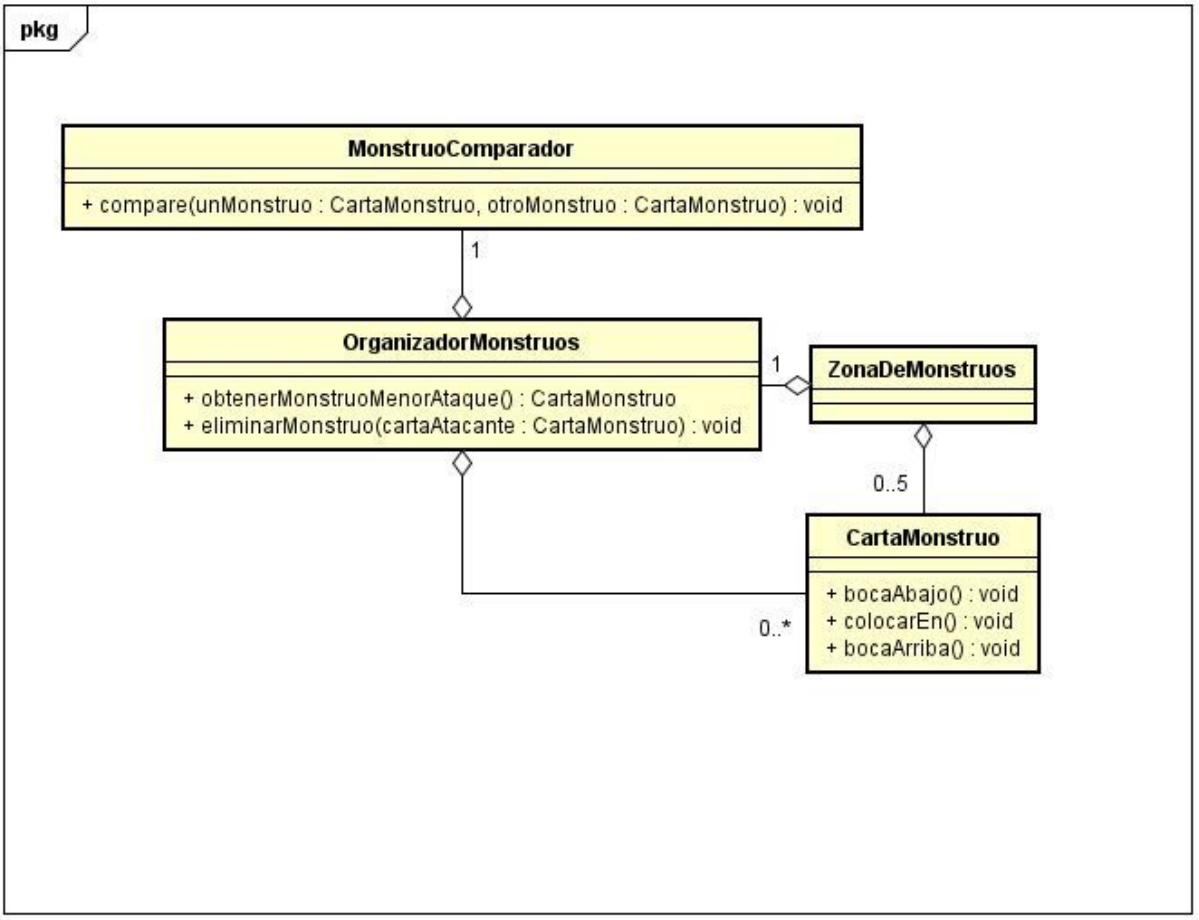


Diagrama de la Zona de Monstruos.

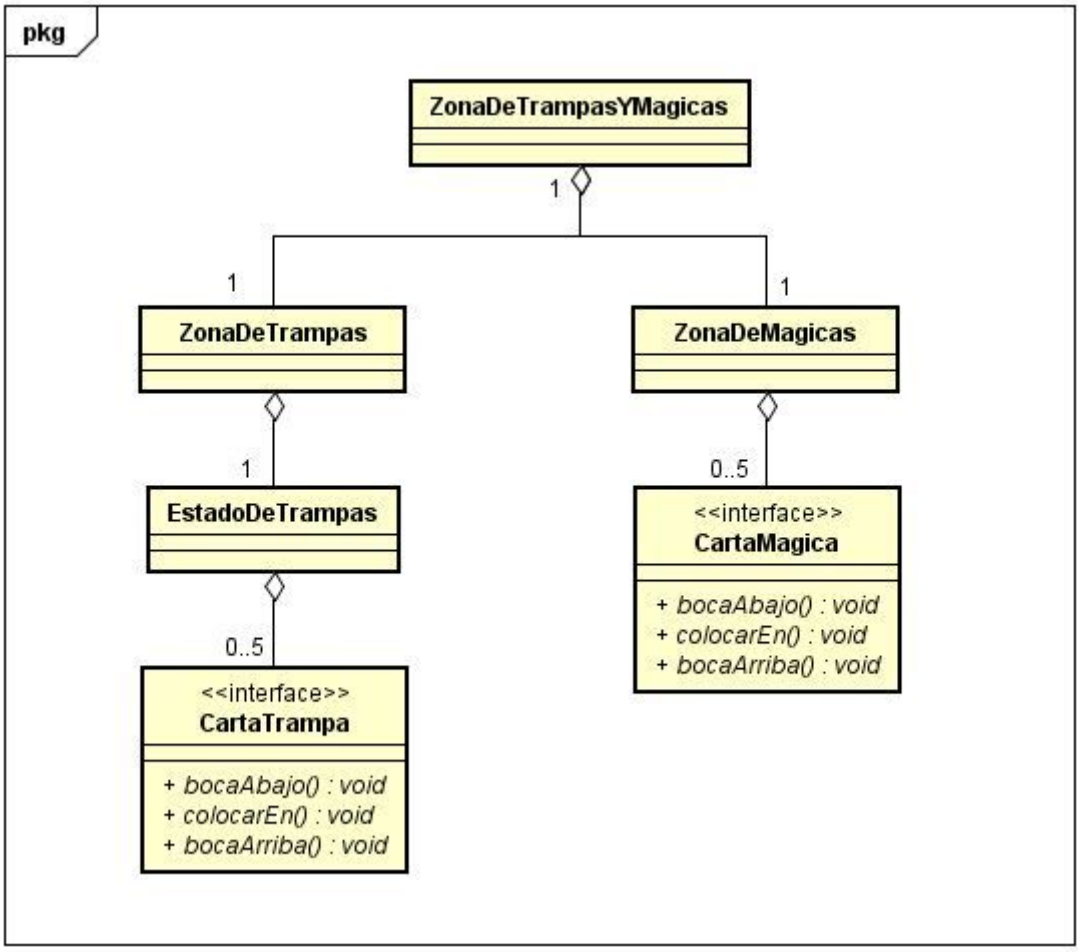


Diagrama de clases de la zona de cartas mágicas y de trampa.

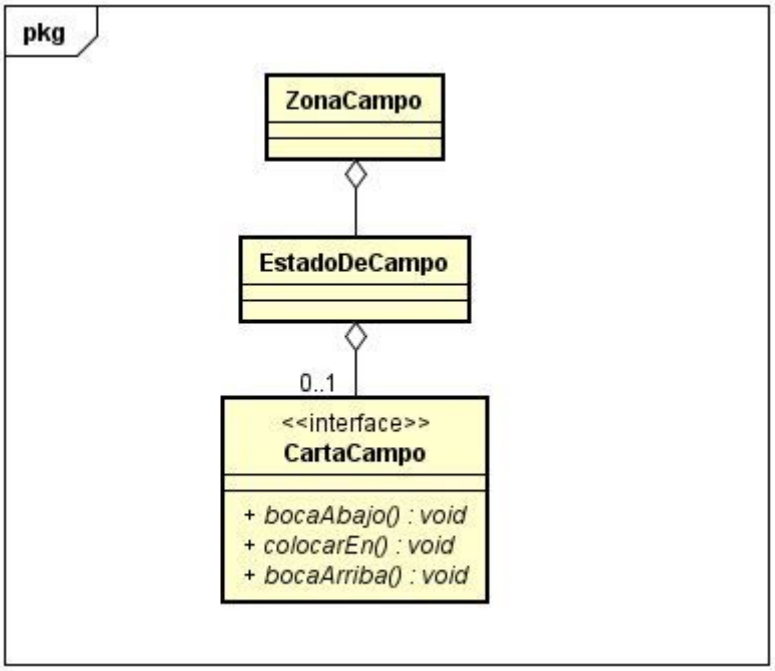


Diagrama de la zona de cartas de campo.

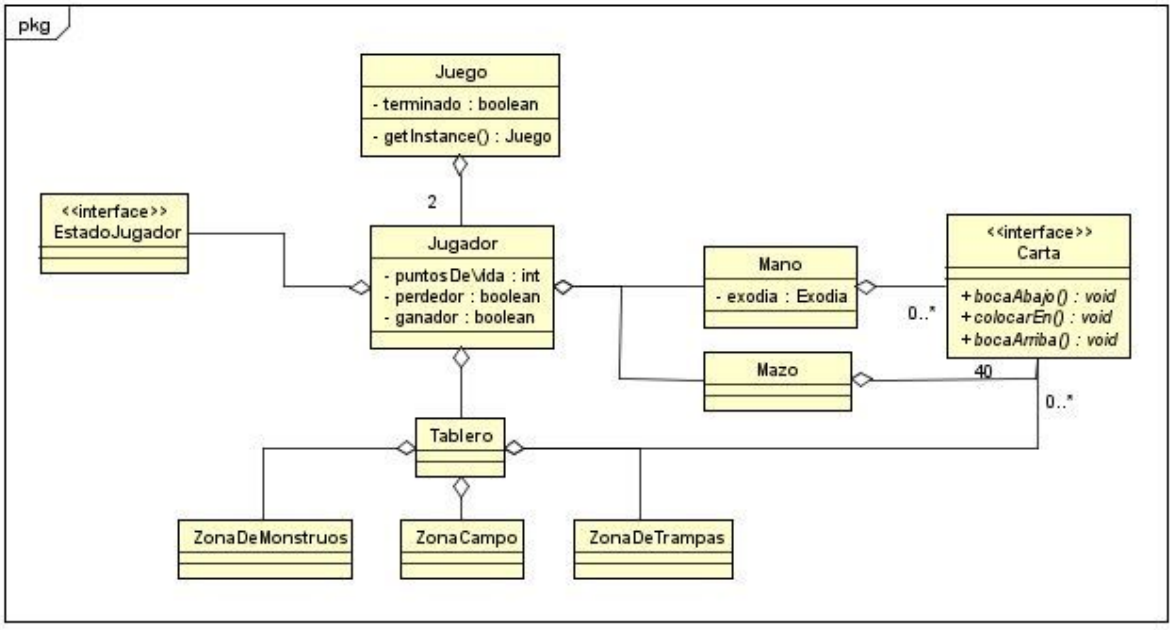


Diagrama de clase general del juego.

Diagramas de secuencia

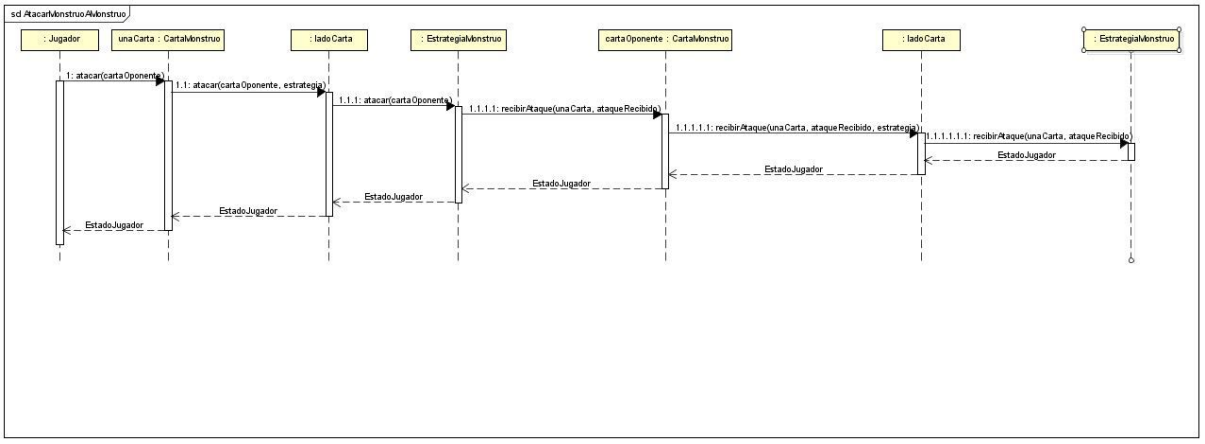


Diagrama de secuencia del ataque de un monstruo a otro monstruo.

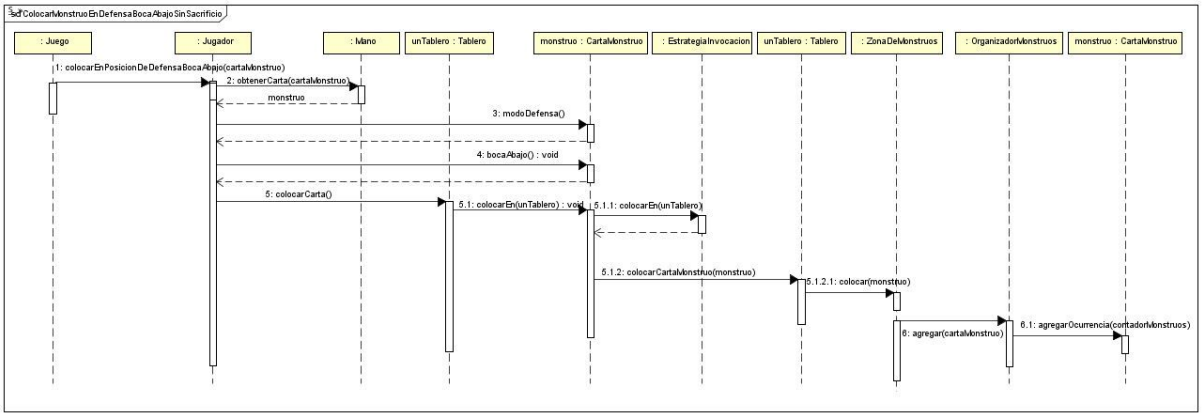
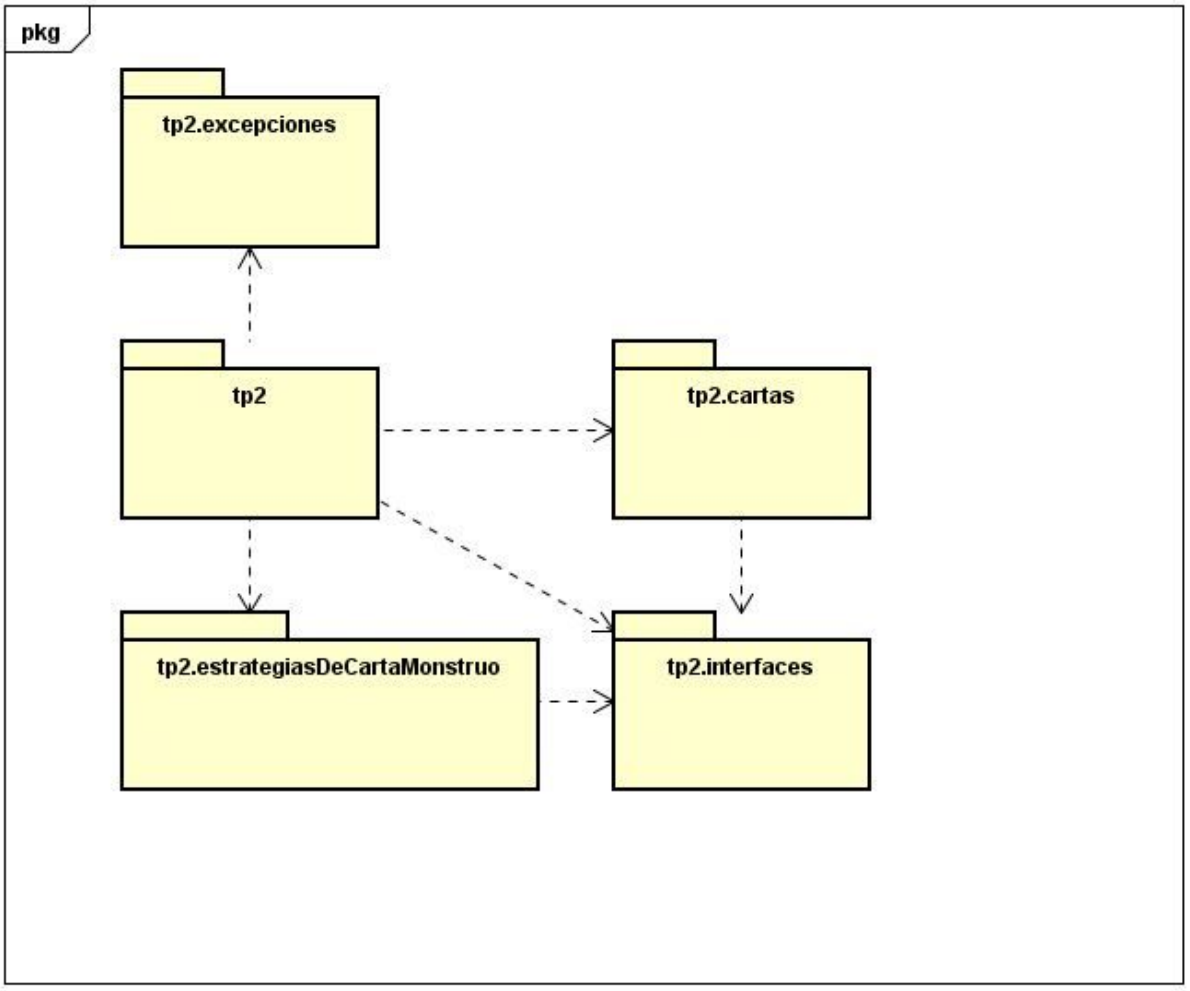


Diagrama de secuencia de la colocación de una carta boca abajo en posición de defensa, sin necesidad de hacer sacrificios.

Diagrama de paquetes



El paquete tp2 el cual contiene las clases que manipulan el juego generalmente, las cuales implementan las distintas interfaces y estrategias.

Diagramas de estado

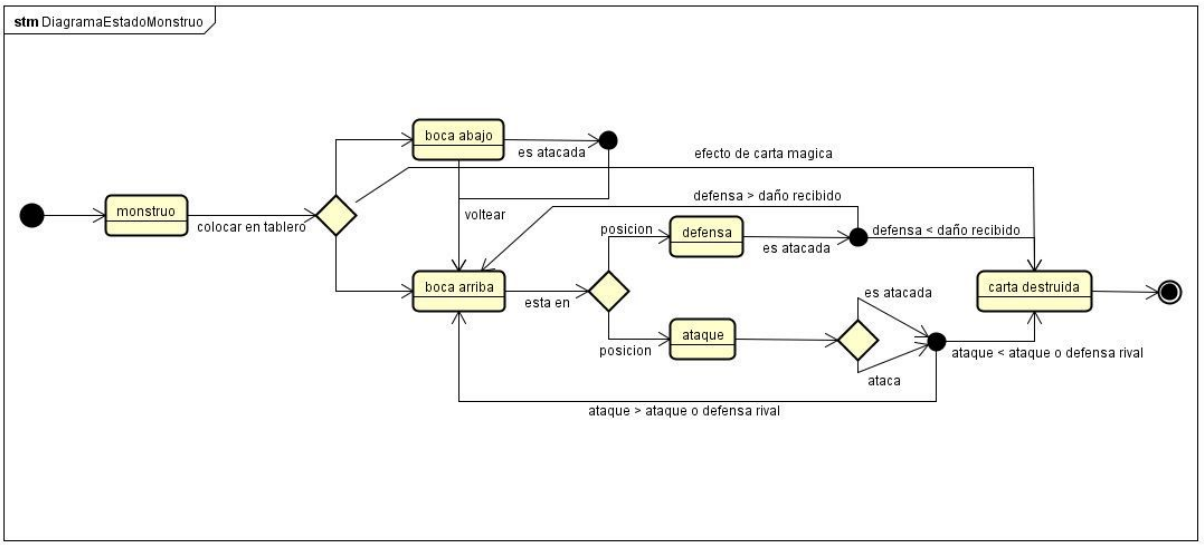


Diagrama de los estados por los que pasa una carta monstruo.

Detalles de implementación

A continuación se detallan ciertas problemáticas que fueron encontrándose a medida que avanzaba el desarrollo del trabajo. Se intentó en la medida de lo posible respetar al máximo el paradigma de orientación a objetos, aplicando el concepto general de objetos enviando mensajes a otros objetos con el objetivo de llegar a la resolución deseada.

La primera dificultad que se presentó a la hora de implementar el modelo es el interrogante sobre la situación del Juego. El Juego es único, y según nuestro modelo nunca existirían dos Juegos en el mismo contexto. Es por esto que se tomó la decisión de utilizar el patrón Singleton, teniendo una única instancia de juego para todo el sistema (accesible desde un método estático de la clase llamado getInstance()). De esta manera el juego puede ser notificado directamente ante situaciones que requieran su atención, como por ejemplo la finalización del mismo como consecuencia de la resolución de un ataque.

Otro desafío encontrado en las fases tempranas de desarrollo fue el de las múltiples combinaciones que puede tener el colocar una carta en el tablero, y las distintas reglas de negocio aplicadas a estas combinaciones. Esta misma dificultad se presentó a la hora de definir la implementación de la invocación de los distintos monstruos. Las distintas variedades son una combinación del modo de ataque (ataque o defensa) junto con la decisión de colocar la carta boca arriba o boca abajo en el tablero. Para poder salvar estas dificultades, y en pos de tener un desarrollo lo más orientado a objetos posible, este desafío fue

resuelto mediante la implementación del patrón Strategy. Logramos así definir la estrategia de ataque/defensa en tiempo de ejecución, dependiendo la misma de la decisión del jugador de colocar una carta en el tablero en una u otra combinación.

De esta manera podemos abstraer complejidad, sacándola de la clase CartaMonstruo para poder trasladarla a las distintas implementaciones de Strategy: EstrategiaLadoCarta, EstrategiaMonstruo y EstrategiaInvocacion.

Asimismo, a la hora de desarrollar los objetos que representarían los distintos lugares del tablero donde pueden ser colocadas las cartas (llamadas Zonas en nuestro modelo), nos encontramos con la prerrogativa que estas zonas modifican su comportamiento según el estado interno en el que se encuentran. Por ejemplo, una zona sin cartas (vacía) puede siempre recibir una carta para ubicarla en la zona, pero una zona ocupada al máximo no podría realizar esa misma operación. Por ello nos pareció funcional a la resolución implementar un patrón State a la hora de encarar esta problemática, teniendo múltiples estados con una interfaz en común (por ejemplo, ZonaTrampasVacia o ZonaTrampasOcupada) resolviendo internamente los cambios de estado sin involucrar a la clase quien los consume. En el caso del ejemplo, quien consume estos estados sería la clase ZonaDeTrampasYMagicas.

Otro punto a nuestro ver conflictivo era el cómo resolver las distintas consecuencias que pueden surgir de un ataque de una carta a otra carta. Observamos que por cada ataque se pueden desprender una cantidad variable de ramificaciones según el resultado del duelo entre dos cartas. Es por esto que la estrategia de resolución que hizo más sentido fue la de Double Dispatch. Encarando la problemática de esta forma, se pudo solucionar el problema mediante llamadas a funciones que dependen del tipo de un objeto específico, pudiendo tener diferentes implementaciones de la misma interfaz que llaman a su vez a distintos métodos según lo que estas implementaciones representan. Por ejemplo, recibir un ataque en “modo defensa” requiere llamadas a distintos métodos que recibir un ataque en “modo ataque”.

Por cuestiones de simplificación del modelo, y dado el alcance del desarrollo, se tomó la decisión (ya mencionada en los supuestos de este mismo informe) de que el jugador no pueda decidir qué monstruos sacrificar cuando invoca un monstruo que requiere sacrificios en su lado del campo. Esto nos llevó a encontrarnos con dificultades para resolver el caso especial del Dragón Definitivo de Ojos Azules, una carta que requiere sacrificios específicos para poder ingresar al campo. Este situación se resolvió mediante la aplicación de un caso especial de invocación para esa carta. La razón principal por la cual este problema fue encarado de esta manera fue principalmente por cuestiones de tiempo. Cabe aclarar que si se desarrollara la funcionalidad de permitir al jugador elegir qué cartas sacrificar, no se requeriría una implementación específica para el Dragon de Ojos Azules.

Otro caso de funcionalidad que no pudo ser completada por cuestiones de tiempo es la refactorización necesaria para poder centralizar métodos que corresponden a todas las cartas, como pueden ser por ejemplo el colocar cartas mágicas o de campo en el tablero, o los métodos para obtener el nombre, que es el identificador de la carta. A la hora de realizar el desarrollo para las entregas iniciales, se tomó la decisión de implementar una clase distinta por cada carta que fuera agregada al mazo de cartas de los jugadores, ya que pareció lo más natural para poder desarrollar evolutivamente y además integrar fácilmente el trabajo de los distintos integrantes del equipo. La consecuencia principal de esta decisión fue que, hacia el final del trabajo y a la hora de pensar una interfaz gráfica, resultó muy difícil agregar nuevos métodos que le correspondieran a todas las cartas, debido a la cantidad creciente de archivos que se debían tocar ante cada modificación.

Es posible que con un poco más de tiempo de desarrollo, esta refactorización pudiera haberse realizado. A fines de poder realizar una entrega funcional y evitar grandes modificaciones que puedan afectar la calidad general de la entrega, se decidió mantener el código como está y añadir esta nota en el informe, para aclarar que la decisión fue evaluada y contemplada a la hora de presentar el trabajo práctico.

Excepciones

A continuación listamos las excepciones lanzadas más comunes en el sistema desarrollado para el trabajo práctico. Cabe aclarar que todas estas excepciones lanzadas heredan de la clase de Java RuntimeException, ya que la intención es que se lancen en tiempo de ejecución y puedan "burbujear" hacia arriba hasta ser atrapadas por algún método.

Estas excepciones se pensaron con la idea de que quien se encargue de atraparlas sea la GUI, pudiendo atrapar excepciones específicas con la intención de mostrarle al usuario mensajes detallados sobre el por qué no puede realizar ciertas acciones.

- **CartaBocaAbajoNoPuedeAtacarException**: Excepción lanzada por la clase LadoBocaAbajo, cuando se intenta atacar con una carta que no está habilitada para hacerlo. Como solo se sabe si esa carta puede atacar o no en tiempo de ejecución, ante los casos que no lo permiten se lanza esta excepción.
- **CartaEnModoDefensaNoPuedeAtacarException**: La explicación es similar a la excepción anterior, con la diferencia que, como su nombre indica, esta excepción es lanzada solo cuando se intenta atacar con una carta en modo defensa.

- **ElCampoEstaVacioException**: Excepción lanzada cuando se intenta obtener una carta de una zona de campo cuyo estado actual es "vacío" (sin cartas colocadas)
- **UnaCartaBocaArribaNoPuedeVoltearseException**: Regla de negocio que establece que una carta colocada en posición Boca Arriba en el campo no puede ser elegida para voltearse a una posición Boca Abajo.
- **LaCartaNoEstaEnElCampoException**: Una carta no puede ser enviada al cementerio, atacar/ser atacada, o activarse su efecto si no fue colocada primero en la zona del campo correspondiente.
- **LaCartaNoEstaEnLaManoException**: Similar a la excepción anterior pero para los casos donde se intenta acceder a la carta desde la mano del jugador.
- **LaZonaDeMonstruosEstaLlenaException**: Esta excepción ocurre cuando se intenta colocar a un monstruo en el tablero sin que haya espacio en el mismo para ubicarlo. El máximo de monstruos que pueden coexistir en la zona de monstruos es 5 (cinco).
- **NoHaySuficientesSacrificiosException**: Excepción lanzada cuando una estrategia de invocación requiere una cantidad de sacrificios mayor a la cantidad de monstruos colocados en el tablero.