

**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** |
| Bellaera, Leonardo | 100973 | leobellaera@gmail.com |
| Gómez Hobbs, Fernando | 94811 | fgomezhobbs@gmail.com |
| Kler, Alejandro | 100596 | alejandrokler@gmail.com |
| Rodríguez Peluffo, Pedro | 99883 | ppedro.rod@gmail.com |

***Fecha de entrega final***: 06/07/2018

***Tutor***:Diego Sánchez

***Comentarios***:

* **Supuestos:**
  + Consideramos que un Monstruo en Modo Defensa no puede realizar un ataque. No está aclarado en la consigna explícitamente.
  + Por default, los monstruos se inicializan en Modo Defensa. Puede cambiarse apenas se crea llamando al método colocarEnModoAtaque().
  + Las cartas se inicializan BocaAbajo por defecto, esto también nos facilita no aplicar efectos de volteo en momentos incorrectos.
  + Si un Monstruo ataca a otro Monstruo que se encuentre en Modo Defensa, éste último no infligirá daño a su Jugador dueño pero sí puede infligir daño al Jugador atacante (por estar en Modo Ataque).
  + Cuando se seleccionan los Monstruos a sacrificar, se evalúa solamente que la cantidad sea la necesaria o que estén los Monstruos correctos. Si esto se cumple pero hay más Monstruos seleccionados, se sacrificarán los necesarios y se destruirán los demás.
* **Modelo de dominio:**

*AlGoOh:* Utilizo la clase como interfaz entre el usuario y el modelo, pensándola como la capa exterior del programa que tiene la responsabilidad de realizar un intercambio directo de mensajes (sin tener en cuenta ninguna interfaz gráfica o menú con facilidades) que permitirá al jugador realizar acciones sobre el juego cuando corresponda.

*Fase:* Es la clase que se encarga de permitir al Jugador realizar acciones según sea el momento.

*Jugador:* Representa a uno de los jugadores que interactuará con el programa permitiéndole realizar cambios en el juego, y tendrá su tablero con zonas y su mano con cartas. Conociendo a su oponente, realizará ataques en su tablero.

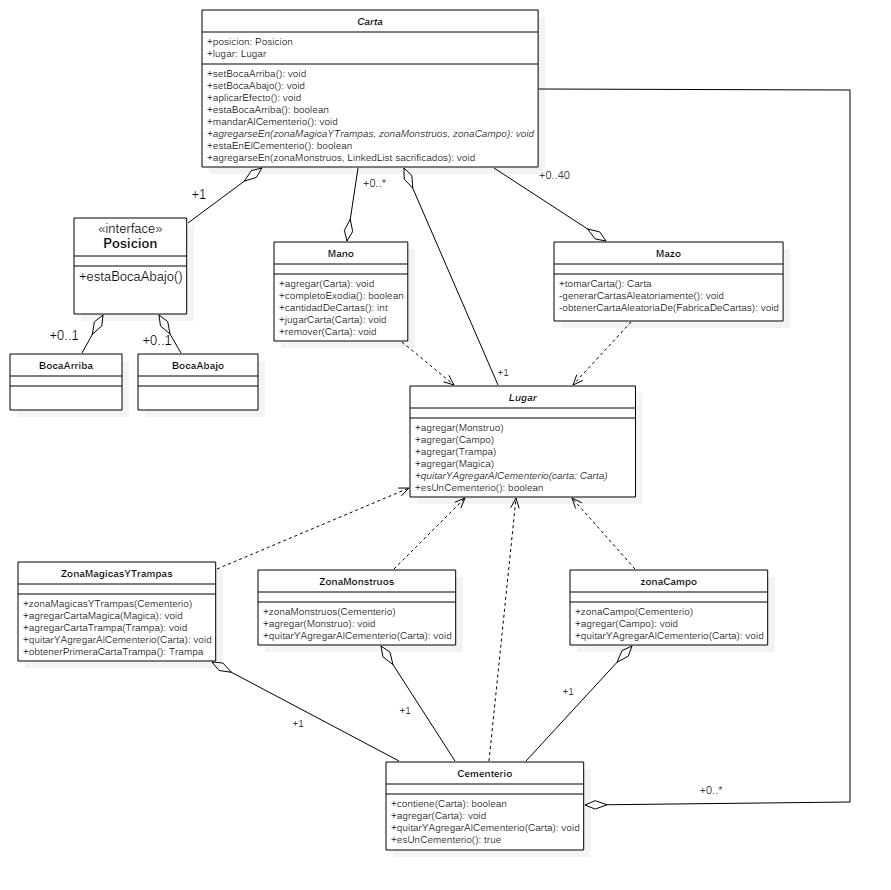
*Monstruo:* Es una clase abstracta que define algunos métodos comunes a todos los monstruos y permite a sus subclases sobrecargarlos para realizar las acciones de manera distinta según corresponda a su comportamiento. Al estar colocadas en ZonaMonstruos, pueden atacar a otros Monstruos en la FaseAtaque y destruirlos, dañando o no al Jugador que lo posea.

*Magica:* Es una clase abstracta que funciona de igual manera que la anterior, declarando métodos para cartas de tipo mágico. Se pueden aplicar en la FaseMagicas para causar efectos que dependen de cada carta.

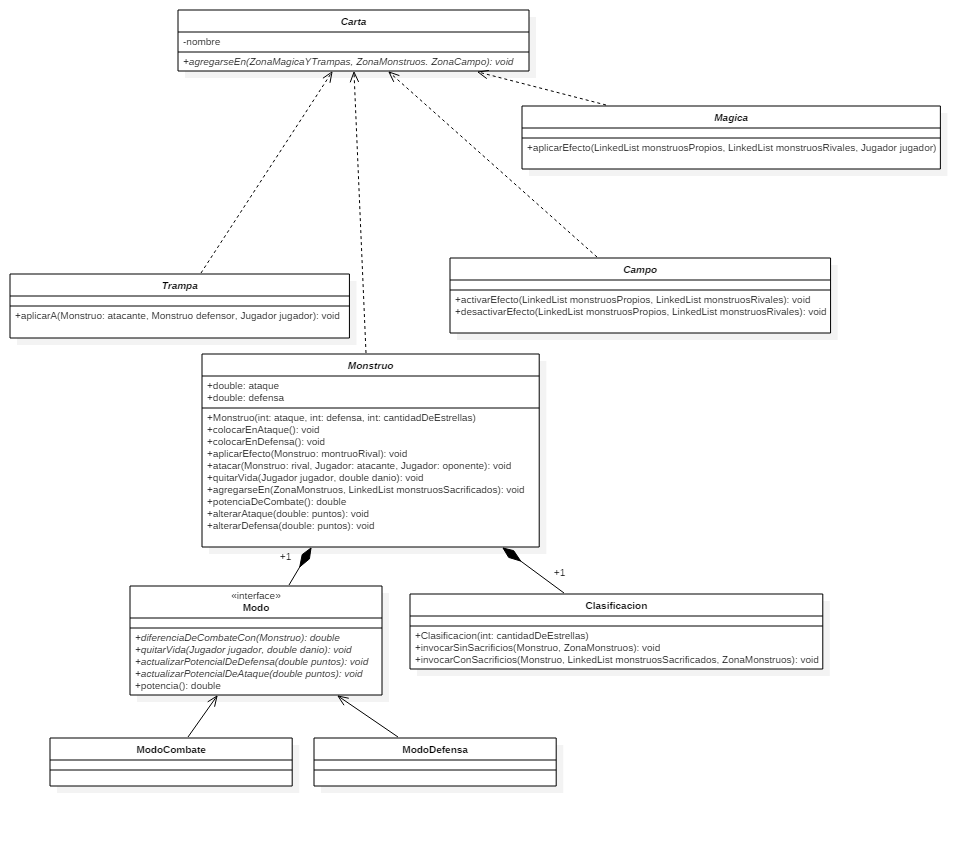
*Trampa:* Es, también, una clase abstracta que declara métodos para cartas de tipo trampa. Si se juegan en la ZonaMagicasYTrampas de un Jugador, serán seleccionadas en orden para aplicarse en los ataques de un Monstruo a otro. Estas primero aplican su efecto trampa y luego, según corresponda, dan lugar al ataque o no.

*Campo:* También clase abstracta, si es jugada en la ZonaCampo puede aplicar su efecto previo a un ataque para modificar los valores con que se producirá el enfrentamiento. Una vez realizado, se desactiva. De esta forma, evitamos aplicar Campo a las cartas que se juegan durante la FasePreparacion y simplemente aplicamos su efecto cuando es necesario. Además, evita que ante un cambio de Campo la carta continúe con el efecto anterior.

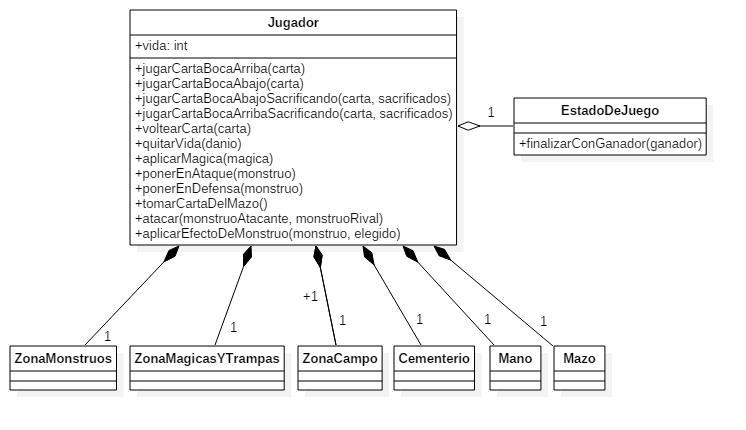
* **Diagramas de clases:**



*Diagrama de las clases relacionadas con Carta y las zonas donde puede estar (Lugar).*



*Diagrama de las clases que modelan cada tipo de Carta.*



*Diagrama de la clase Jugador y las zonas donde mueve sus cartas. Tiene una referencia al EstadoDeJuego para terminar el juego cuando corresponde.*

* **Detalles de implementación:**

Utilizamos los patrones Null Pattern, Strategy y State para modelar distintos comportamientos de una misma clase.

Utilizamos el patrón Double Dispatch para métodos que realizaban una misma tarea con una pequeña diferencia en los parámetros necesarios.

Utilizamos los patrones MVC y Observer para el diseño de la interfaz gráfica.

Utilizamos Factory Method para la creación de cartas y las tomamos cuasi-aleatoriamente para generar un Mazo.

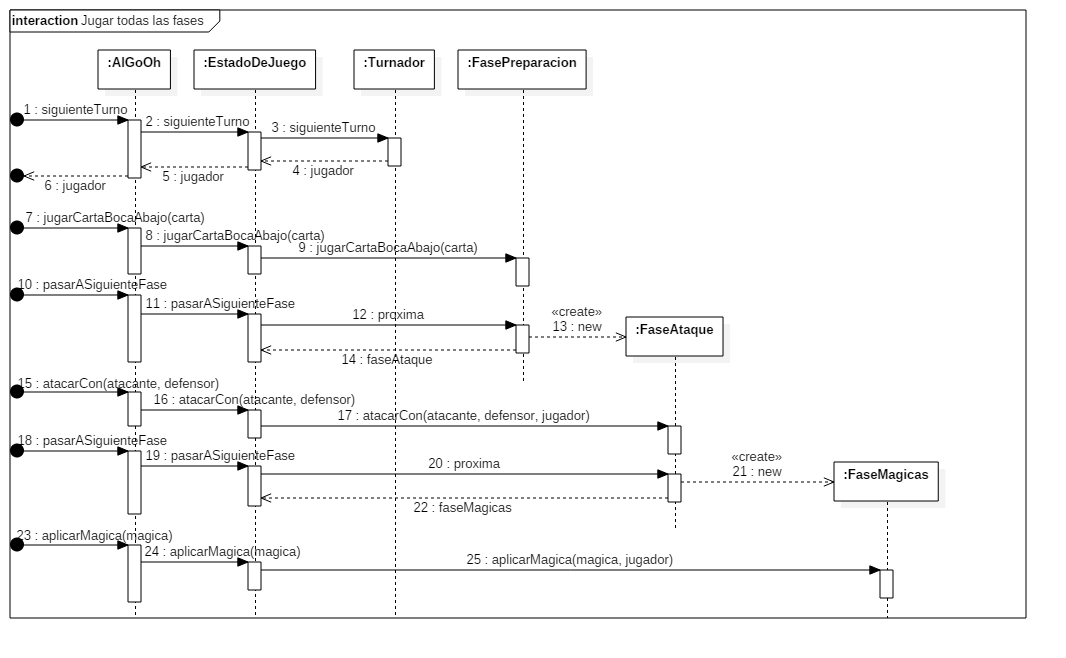
No le delegamos la responsabilidad de atacar con Monstruos y Trampas al Campo ya que son 2 Campos los que entran en juego y deben activarse para distintos Jugadores y Monstruos, y luego desactivarse.

A Monstruo le paso Jugador (y no un objeto Vida) para notificar al Jugador que está siendo atacado y éste poder cambiar el EstadoDeJuego a Terminado declarando ganador al oponente (finalizarConGanador(ganador)).

AlGoOh se inicializa con un turno al azar sin ser solicitado (sin llamar al método siguienteJugador()) para evitar NullPointers.

* **Excepciones:**

1. *ExcepcionProductoNoExiste:* Creada para dar aviso cuando un producto no está registrado en el inventario de AlgoBay. Al intentar agregar un producto inexistente a una compra mi implementación no realizará ninguna acción en el programa. Esta excepción puede lanzarse en los métodos agregar:aCompra: y getProducto: de la clase AlgoBay.
2. *ExcepcionCompraNoExiste:* Similar a la anterior, es creada para advertir de una compra que no figura en el listado. Da aviso cuando un producto o un cupón se quieren añadir a una compra no registrada, ya que de no conocerla, mi implementación no realizará acción alguna sobre esta en el bloque de código llamado. Puede lanzarse en los métodos agregar:aCompra:, agregarCupon:enCompra: y getPrecioTotalDe: de la clase AlgoBay.
3. ExcepcionCuponNoExiste: Puede arrojarse en la clase ServicioExtra cuando se busca el cupón aplicado y se atrapa en la clase Compra respondiendo de manera que solamente agregue el cupón nuevo. Claramente, su función es dar aviso de que no hay un cupón previamente aplicado a la compra.
4. *ExcepcionPorcentajeInvalido:* Creada para evitar que se introduzcan porcentajes menores a cero o mayores a cien en un cupón, ya que no tendría sentido el descuento que aplica según el contrato. Puede arrojarse en el método setPorcentaje: de la clase Cupon.
5. *ExcepcionPrecioNegativo:* Esta excepción no permite el ingreso de cantidades negativas para asignar el precio a un producto y por ende puede aparecer al llamar incorrectamente al método setPrecio: de la clase Producto.
6. *NotFound:* En los métodos en los cuales utilizo detect: para una OrderedCollection es posible que no encuentre el objeto buscado, por lo que arroja la excepción NotFound y la manejo de modo que devuelva nil. Esto se da en ServicioExtra donde puede que se realice una compra sin envío, sin garantía, o sin cupones y la lista de servicios se encuentre vacía. También puede darse cuando busco las compras en el listado o los productos en el inventario, de manera que mi implementación arroja la excepción correspondiente a cada caso.

* **Diagramas de secuencias:**

*Para comenzar un turno, se llama al método* ***siguienteTurno()*** *que devuelve el Jugador al cual le toca jugar y queda almacenado en EstadoDeJuego.*

*Para avanzar en las fases, se llama al método* ***pasarASiguienteFase()*** *que permite al Jugador realizar determinadas acciones que correspondan a la Fase próxima.*

*Para llamar métodos del jugador, se pueden utilizar* ***varios métodos de AlGoOh*** *que a través del EstadoDeJuego y sus Fases controlará en qué momento se pueden utilizar.*

*De no ser posible realizar una acción durante determinada Fase, se lanzará ExcepcionFaseIncorrecta.*