

**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().
  + 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.

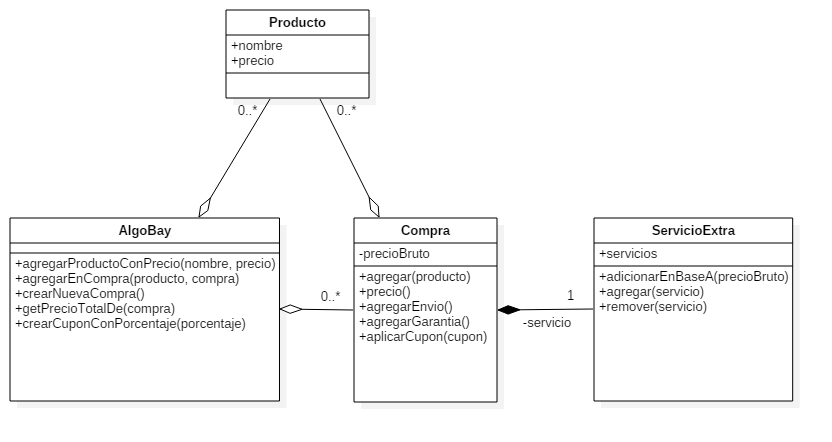
*Monstruo:* Es una clase abstracta que define algunos métodos comunes a todos los monstruos y permite a sus subclases sobrecargarlos para realizar las acciones de manera distinta según corresponda a su comportamiento.

*Garantia:* Se añade como Servicio a la Compra correspondiente según tenga garantía con el fin de sumarse al precio total.

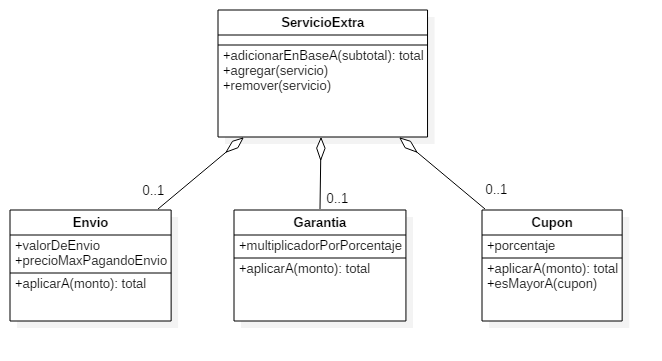
*Cupon:* Se añade por último a una Compra para aplicar el descuento que corresponda según su porcentaje. Prevalece el de mayor porcentaje. Sabe comparar su propio porcentaje con otro. Si bien implementa la interfaz ServicioExtra, el cupón es tratado por su clase en todo momento, ya que no se encuentra dentro de una lista de servicios (como lo hice en el TP1-S).

*Producto:* Representa un producto con nombre y precio. Su precio no puede ser negativo. Su única responsabilidad es brindar acceso a sus atributos.

* **Diagramas de clases:**

****

*Diagrama de las clases principales.*



*Diagrama de las clases que representan un servicio adicional.*

* **Detalles de implementación:**
* **Utilizamos los patrones Null Pattern, Strategy, State y Double Dispatch.**
* **Utilizamos Factory Method para la creación de cartas y las tomamos aleatoriamente para generar un Mazo**
* **No le delegamos la responsabilidad de atacar con Monstruos y Trampas a 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.**

Al pensarlo como un sistema de comercio electrónico similar a la realidad, mi modelo almacena un listado de inventario que registra y almacena todo agregado de productos. Esto es también una suposición que hice, pero no detallé en la sección de supuestos para no hablar de la implementación.

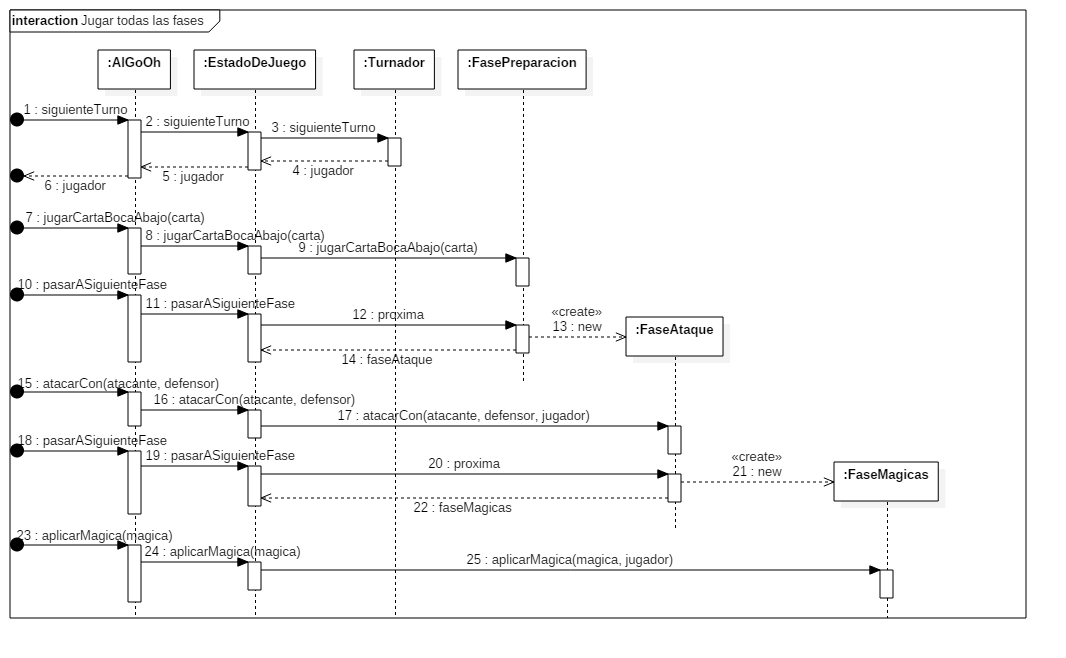
Respecto a la diferencia entre las compras con o sin envío y/o garantía, aprovecho los conceptos de polimorfismo para evitar utilizar herencia en clases como “CompraConEnvio”. Para ello, utilizando una interfaz, envío el mensaje aplicarA: a cada servicio (envío, garantía; cupón ya no) del ServicioExtra. Cada clase lo interpreta a su manera y devuelve un resultado. De esta forma, los servicios que estén aplicados aportarán su tarea y si no hay servicio alguno no habrá cambios en el subtotal.

Para la clase Envio establecí su valor y el precio mínimo sin pagarlo [el envío] (VALOR\_MAX\_PAGANDO\_ENVIO) como atributos constantes de la clase para que en caso de modificación de los números no hubiera que retocar el código.

Lo mismo aplica para la clase Garantia, donde su multiplicador por porcentaje y el mismo porcentaje aparecen como atributo constante.

* **Excepciones:**
* *Nota: Tuve que sacar de mi modelo las excepciones 1, 2, 3 y 6 para evitar modificar las pruebas dadas por los docentes. Sólo utilizo las excepciones 4 y 5.*

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.*