# FIUBA - 75.07 - 95.02

# Algoritmos y programación III

*Trabajo práctico 2: AlgoMon*

2do cuatrimestre 2016

(trabajo grupal de 3 o 4 integrantes)


Alumnos:

| Nombre | Padrón | Mail |
|---|---|---|
| Calarota, Marcelo Luis | 94839 | mlcalarota@gmail.com |
| Fonseca, Matías | 98591 | fonseca.196@gmail.com |
| Horn, Miguel Agustin | 98124 | agushorn145@gmail.com |
| Zambelli Tello, Brian | 98541 | brianzambellitello@gmail.com |


***Fecha de entrega final***: Jueves 1/12/2016 - acordar con ayudantes.

***Tutor***: Degiovannini, Marcio

***Comentarios***:

# Informe

## Supuestos

- Luego de cada acción que realice un jugador, ya sea atacar, cambiar de algomon, o usar un elemento, pasa de turno correspondiéndole el mismo al jugador oponente. Asimismo, cuando un algomon muere, el jugador debe elegir el siguiente algomon activo. Después de realizar esta acción, también pasa de turno.

- En la segunda entrega, el punto: *“2. Bulbasaur ataca con chupavidas a Charmander y le quita 30 puntos y gana 9 puntos de vida.”* es incorrecto: en realidad le saca 7 y gana 2.

- En la segunda entrega, el punto: *“3. Bulbasaur ataca con chupavidas a Squirtle y le quita 7 puntos y gana 2 puntos de vida.”* es incorrecto: en realidad le saca 30 y gana 9.

## Modelo de dominio

Inicialmente comenzamos realizando un diagrama de clases tentativo para el modelo, poniendo en la misma la clase Algomon y Ataque. Luego para resolver la problemática de los tipos, tanto de los algomones como de los ataques, decidimos crear también la clase Tipo para manejar el comportamiento correspondiente.

Para la resolución de los ataques especiales que dejan con estados particulares a los enemigos, creamos las clases EstadoDormido, EstadoNormal, EstadoQuemado las cuales implementan la interfaz Estado. Cada algomon tiene un atributo estadoEfimero y estadoPersistente.

Finalmente para los ítems, realizamos un modelo análogo al de los estados.

Para manejar los ítems, los algomones, y los ataques de los algomones, creamos la clase Jugador, la cual contiene una lista de algomones, un diccionario de los items disponibles, y un algomon activo, que es del cual el jugador podrá elegir los ataques a realizar, y dicho algomon

será el que reciba el daño del jugador oponente.

Ya teniendo todo el modelo avanzado, decidimos crear la clase Juego la cual tiene los 2 jugadores, y coordina el turno de cada uno. Para simular un juego, solo basta con comunicarse con esta clase, y esta es la encargada de mandar los mensajes correspondientes a otras clases.

Con el modelo completo, empezamos a crear la interfaz gráfica. Inicialmente comenzamos de una manera parecida a cuando empezamos el modelo, pensando las clases básicas que debe tener, basándonos en la librería JavaFx.

La primer cuestión a resolver, fue la de relacionar el modelo con la interfaz gráfica. Para ello, se creó la clase ControladorLogicoDelJuego. Su función, es la de relacionar lo que ingresa el usuario final de la aplicación, por medio de la interfaz, con el modelo en si, para un correcto comportamiento.

Analizando las clases básicas que habíamos pensado, nos dimos cuenta que en su mayoría, estaban compuestas por botones, por lo que una clase capaz de crear los botones que necesitábamos, resultó imperativo.

A medida que avanzamos con el diseño de la interfaz, el código se iba haciendo cada vez más extenso, por lo que surgió la idea de separar el código en clases que representaran las distintas pantallas que se utilizaban. Así, surgieron, PantallaInicial, PantallaEleccion, PantallaBatalla, PantallaVictoria. El desarrollo de cada una de estas pantallas, decantó en la necesidad de distintas clases que suplieran funcionamientos faltantes. La pantalla de elección, significó la creación de un creador de botones específico. La pantalla inicial, la de opciones. Y la pantalla Victoria con MenuTop y la clase Notificaciones, para poder ir notificando de los distintos sucesos de la partida.

En la sección 'Detalles de implementación' explicaremos lo acá dicho de una manera más explicativa.

## Diagramas de clases

Hay 2 diagramas de clases: uno correspondiente al modelo del trabajo, y otro correspondiente a la vista del mismo. Decidimos realizarlos por separado para un mayor entendimiento de los mismos.

## Diagramas de secuencia

Los diagramas se encuentran adjuntos en PDF. Se realizaron dos diagramas de secuencia. El primero muestra cómo ataca un algomon utilizando sólo la parte lógica. Mientras que el segundo diagrama muestra la secuencia realizada por la pantalla para cargar los algomones elegidos por cada jugador.

## Diagrama de paquetes

El diagrama se encuentra adjunto en PDF. Podemos observar la relación de todos los paquetes dentro de nuestro proyecto y cómo se comunican entre ellos.

## Diagramas de estado

Se realizaron 2 diagramas de estados: uno correspondiente al estado de un Algomon a lo largo del juego, y otro correspondiente al estado del juego durante su ejecución.

## Detalles de implementación

En un principio, encontramos inconvenientes a la hora de empezar a realizar el trabajo práctico. No sabíamos si realizar una clase genérica Algomon o crear una clase abstracta Algomon y de ella que hereden los 6 Algomones que se nos asignaron. Debido a este interrogante, consultamos con nuestro tutor y nos dio la razón al decirle que optábamos por la segunda

opción, ya que funcionarían como molde para todos los Algomones que creemos. Esto se debe a que nada más se pueden crear esos Algomones y en principio no cambiarían en el futuro.

Para resolver el problema de los Tipos de Algomón y los ataques, que a su vez éstos últimos también tenían tipo, utilizamos un patrón llamado *Double Dispatch* en la interfaz Tipo. Así cada tipo tiene una respuesta a otro tipo y poder calcular el verdadero daño que causa cada ataque al Algomón contrario.

Mientras que para los ítems creamos una interfaz llamada Item solamente con tres métodos, que las cuatro clases que la impementan (Poción, Super Poción, Vitamina, Restaurador), pueden redefinir estos métodos debido a que cada una realiza acciones distintas.

Cada Algomón utiliza tres clases de Estados (Estado Normal, Estado Dormido y Estado Quemado) para poder modelar los efectos que pueden producir el ataque además de restar vida. Para ello utilizamos la interfáz Estado que funciona igual que Item.

Para terminar con la lógica del juego, realizamos la clase Jugador y Juego. El Juego es el organizador de todas las cosas y a su vez, tiene como atributos 2 instancias de la clase Jugador. Cada Jugador va a poder tener 3 Algomones y una cierta cantidad de items.

Luego procedimos a realizar la interfáz gráfica. Esto fue una gran novedad, debido a que ningún integrante del grupo había hecho alguna. Fue entonces cuando empezamos a investigar sobre JavaFx y comenzamos con la interfaz. Para llevar a cabo el patrón MVC tuvimos que crear un ControladorLógicoDelJuego. Mediante esta clase comunicábamos la vista con la parte lógica.

Para la vista, intentamos realizar un flujo de pantalla. Sin embargo, debido a la falta de experiencia y nuestro conocimiento básico de interfaz gráfica, pudimos realizar un flujo principiante de 3 pantallas. Cada pantalla es una clase distinta en donde se pasa por parámetro el Stage y vamos cargándole distintas escenas que representa cada pantalla. En cada una de estas, para que no quede muy extenso, poco legible y entendible, definimos métodos genéricos que utilizan los botones a la hora de accionarlos, lo que emprolijó el código y lo aprovechamos para no repetir código.

Además se crearon clases en la vista que nos ayudaron a la hora de

encapsular cosas, como CreadorBotón o MenuTop o el ReproductorDeSonidos. En este último, realizamos un patrón *State* que nos ayudó a la hora de disponer de los distintos sonidos y música en el programa.

## Excepciones

Creamos 5 excepciones en todo el desarrollo del trabajo práctico. A continuación se explicará la finalidad de cada una y que hacemos cuando se la captura:

- *EstaDormidoException*: la misma se lanza para indicar que un algomon está dormido. La misma se captura cuando un algomon dormido quiere atacar, y se debe esperar a realizar el cuarto ataque para despertar.
- *SinPuntosDePoderException*: la misma se lanza para indicar que no quedan puntos de poder para que un algomon realice cierto ataque. La misma se captura cuando se realiza el último ataque disponible, dejando imposibilitada la opción de elegir dicho ataque.
- *SinUsosDisponiblesException*: la misma se lanza para indicar que no quedan usos disponibles de cierto ítem. La misma se captura cuando se realiza el último uso de cierto ítem, dejando imposibilitada la opción de elegir nuevamente ese ítem.
- *PokemonMuertoException*: la misma se lanza para indicar que un algomon tiene vida menor o igual a cero. La misma se captura cuando un algomon recibe un ataque que lo deja con la vida igual o menor a cero, y en ese momento se le avisa al jugador que debe cambiar de algomon. El algomon que lanzó la excepción queda imposibilitado de ser elegido.
- *VictoriaObtenidaException*: la misma se lanza para indicar que un jugador es el ganador. La misma se captura cuando un jugador tiene 2 algomones muertos, y el algomon que tiene activo recibe un daño que lo deja con vida menor o igual a 0. En ese instante se termina la partida.