*75.07 – Algoritmos y Programación III*

**Trabajo Práctico 2:**

**Juego de estrategia por turnos**

1°C 2017 – Trabajo grupal

**Fecha de entrega:** 29-06-2017

**Corrector:** Pablo Suarez

**Integrantes:** Sportelli Castro Luciano – 99565 – sportelliluciano@gmail.com

Berrotarán Pablo – 98446 – [berrotaran.p@gmail.com](mailto:berrotaran.p@gmail.com)

**Calificación:**

**Introducción**

En el presente informe se detalla el diseño, implementación y desarrollo de la consigna que fue asignada por la cátedra. Se incluyen especificaciones que se hayan considerado importantes; tales como el diseño general del programa; sus motivos y consecuencias; así como los diagramas necesarios para facilitar la comprensión del flujo y estructura. También, se realiza un análisis de las pruebas realizadas y del esquema con el que se trabajó para realizar la interfaz gráfica del programa.

Para el desarrollo del trabajo práctico, se utilizó Java 1.8 en conjunto con JavaFX. Los diagramas presentados fueron realizados mediante el estándar UML.

**Consigna**

Se le asignó al grupo como objetivo del trabajo práctico el realizar el diseño e implementación de un juego de estrategia por turnos[[1]](#footnote-1) con la popular serie de manga y anime Dragon Ball Z[[2]](#footnote-2) acorde a los conceptos dados en clase que prioricen el diseño adaptativo y eficiente y la prolijidad de código, usando las herramientas vistas (Java y JavaFX[[3]](#footnote-3)) para desarrollar un modelo e interfaz gráfica acompañante. El producto final es un juego que un usuario puede ejecutar en su computadora de forma local, incluyendo en el mismo gráficos con los cuales el usuario puede controlar el juego.

Para cumplir con los estándares de calidad de código, así como la funcionalidad del mismo, se trabajó acorde al esquema de desarrollo manejado en pruebas (TDD); que se define como un esquema de desarrollo que software que se basa en la repetición de un ciclo corto de pruebas unitarias, poniendo el foco del proceso de implementación en cumplir con el funcionamiento del ciclo, y emprolijar el código previamente escrito.

Durante el desarrollo de la consigna, se mantuvo el objetivo de maximizar los beneficios inherentes de un esquema en TDD; aprovechando que el mismo promueve el diseño eficiente y poco superfluo.

También se especifica en la consigna la necesidad de implementar clases para completar el programa, siguiendo un esquema de Programación Orientada a Objetos (POO); lo cual implica que es necesario llevar a cabo criterios de clases con los contratos correctos, asegurando que la comunicación entre las clases sea fácil de entender y resuelva los problemas presentados de forma ágil, capaz de integrar fácilmente nuevas funcionalidades que puedan darse en el futuro.

**Organización del equipo**

Siendo este un trabajo de modalidad grupal, se puso énfasis desde un principio en el trabajo en equipo, realizando una distribución equitativa del trabajo que permitiera a cada miembro del equipo hacer el mejor uso posible de sus habilidades y trabajar acorde a sus intereses y motivaciones.

También, se consideró de suma importancia el mantener comunicación constante entre los miembros del equipo para no realizar trabajo superfluo, acorde a los conceptos de integración continua[[4]](#footnote-4) y eXtreme Programming[[5]](#footnote-5). Con este fin, se utilizó un repositorio común en GitHub, y se acordó el realizar integraciones periódicas al código, asegurándose a cada paso que el código que cada miembro había escrito sea compatible con la integración anterior. Llevando a cabo este modo de trabajo, se facilitó la cooperación entre integrantes.

**Desarrollo de la implementación**

Para llevar al cabo el diseño del programa, se consideró primero el rango de funcionalidades requeridas por el juego. A partir de este análisis, es evidente la necesidad de encontrar un manejo eficiente del modelo lógico del programa, de los gráficos incluidos, y de la relación entre los mismos. Para este fin, se separaron los elementos del código en modelo y vista, ocupándose estos de llevar a cabo la lógica interna del juego, y de dar output y tomar input del usuario, respectivamente. En cuanto al modelo, el mismo fue segmentado en clases para hacer una distribución intuitiva de las responsabilidades; comenzando con las clases que representan elementos no tangibles como ser el juego, los jugadores y equipos. El juego progresa comunicándose con el tablero y las unidades, dando las órdenes que ingrese el usuario, y las unidades se comunican entre ellas enviándose mensajes que contienen el efecto que se desea.

Para llevar el diseño a la práctica, se utilizaron prácticas propias de la programación orientada a objetos como ser el uso de patrones de diseño cuando corresponda, y el uso del polimorfismo para producir código versátil, fácil de entender y capaz de adaptarse a modificaciones futuras.

Clases: En la ejecución del programa, la primera clase en tomar protagonismo es la clase ***DragonBallApp***. Esta clase se ocupa de inicializar el programa, conteniendo el método main y dibujando la ventana en la que el juego transcurre. *DragonBallApp* cuenta en su contrato con la capacidad de crear los demás objetos necesarios para inicializar el juego, y poner al usuario en contacto con los mismos mediante la inicialización de la interfaz gráfica.

*DragonBallApp* inicializa un objeto ***Juego***. El objeto *Juego* es el responsable de crear el tablero y registrar la progresión de la partida, agregando periódicamente un consumible al tablero y cambiando en cada turno el jugador a quien corresponde jugar. Cada *Juego* se comunica con dos objetos de clase ***Jugador***, que por su parte tiene comunicación con sus unidades, encargándoles que se muevan o modifiquen sus estados (vida, ki o transformación) según el usuario lo especifique mediante input.

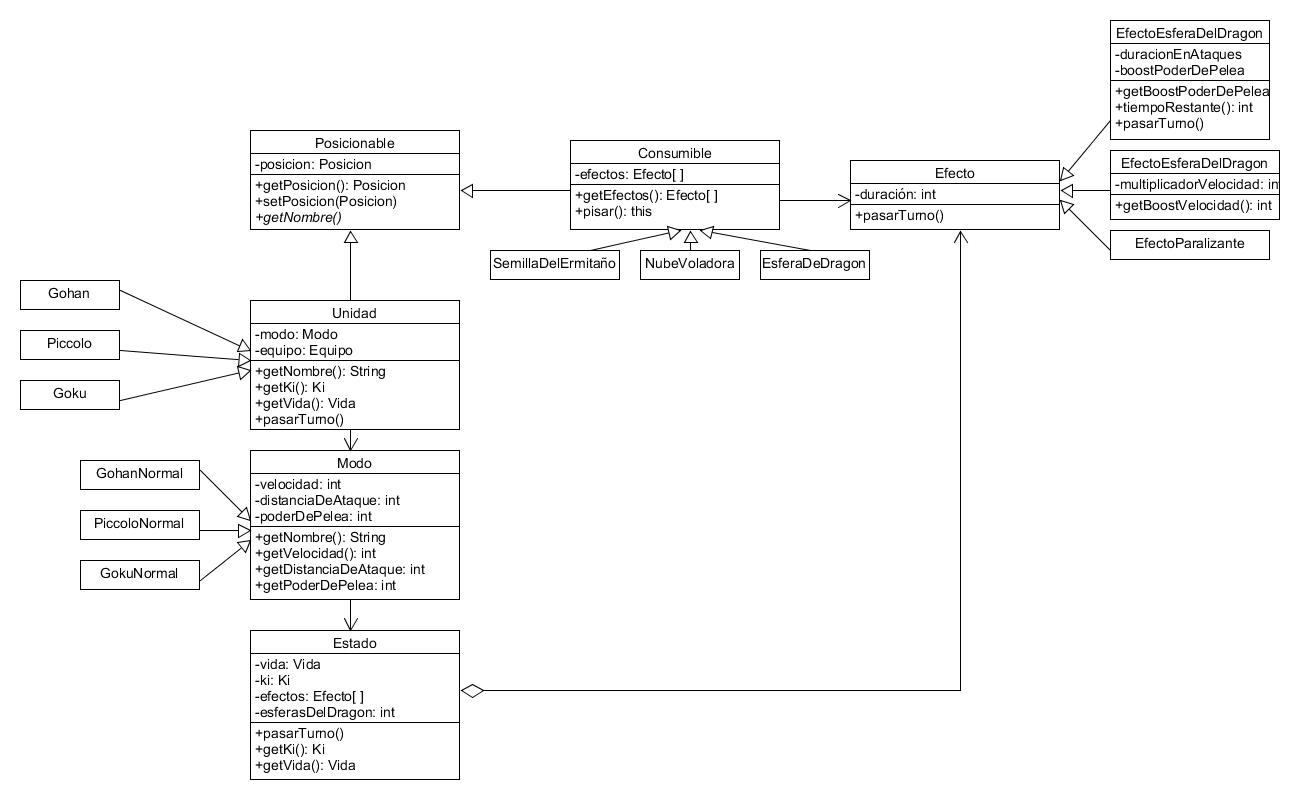
El ***Tablero*** es la clase que representa la grilla de posiciones libres y ocupadas, manteniendo constancia de donde se encuentra cada unidad del juego. Como tal, es el *Tablero* el que conoce si una unidad es capaz de moverse a una posición determinada, efectivamente teniendo que cada unidad delegue al tablero es cálculo de si un movimiento es o no válido.

Por su parte, ***Posicionable*** es una clase abstracta que agrupa a los objetos que pueden ubicarse en la grilla; tanto ***Consumibles,*** posicionables que pueden ser recogidos por una unidad para afectar sus atributos, como ***Unidades***, los personajes que cada jugador maneja y debe utilizar para intentar ganar la partida.

Cada ***Consumible*** cuenta con un ***Efecto.*** El *Efecto* tiene la responsabilidad de aplicarse a los ***Atributos*** de una *Unidad* cuando esto sea necesario. ***Unidad***, por su cuenta, posee un ***Estado;*** aquí se utilizó, para maximizar las capacidades de polimorfismo de cada instancia, un **Patrón de *State*** que delega a esta clase la responsabilidad de mantener constancia de la información sobre una unidad dada. Por su parte, *Estado* divide sus responsabilidades en tres bloques: Una lista de efectos a los cuales la unidad está sujeta, que pueden ser de un consumible o de un ataque especial, un ***Modo*** que mantiene constancia de la vida y el ki; y una ***Transformación*** que dicta el poder de ataque y capacidad de movimiento de la unidad.

En el momento de determinar el daño y capacidad de movimiento, la *Unidad* pide a su *Modo* que acceda a sus ***Atributos*** y le aplique los *Efectos* antes de devolver el atributo específico buscado.

Las clases cuentas con un método pasarTurno() donde aplique, que actualiza sus respectivas variables de forma tal que registre que el jugador ha decidido terminar el turno y, con esto, aumentar el ki de sus unidades y reducir la duración de todos los efectos.

Este comportamiento puede apreciarse mejor en el siguiente diagrama de clases:

**Pruebas**

Con la consigna del trabajo práctico, se incluyeron sets de pruebas de integración. El análisis de las mismas permitió concluir el alcance del programa propuesto y orientar los esfuerzos del equipo en torno a cumplir estos requisitos.

Para asistir con el ciclo de desarrollo del programa, se escribieron también pruebas unitarias complementarias al código. Estas están incluidas en el paquete test de la entrega, y cubren el funcionamiento de las clases de forma unitaria, probando las funcionalidades de cada clase individualmente. En caso de necesitar refactorear o expandir el código, las pruebas unitarias provistas deberían facilitar la comprobación de que se estén cumpliendo los objetivos propuestos.

1. https://es.wikipedia.org/wiki/Videojuego\_de\_estrategia\_por\_turnos [↑](#footnote-ref-1)
2. https://es.wikipedia.org/wiki/Dragon\_Ball [↑](#footnote-ref-2)
3. http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784 [↑](#footnote-ref-3)
4. https://aws.amazon.com/es/devops/continuous-integration/ [↑](#footnote-ref-4)
5. https://es.wikipedia.org/wiki/Programaci%C3%B3n\_extrema [↑](#footnote-ref-5)