**Informe Proyecto POO**

**¿Cómo funciona?**

El juego crea un tablero dependiendo del tamaño escogido al iniciar la partida. Se le otorgan a los dos jugadores una misma cantidad de recursos que se utilizaran para comprar su elección de piezas. El tablero es creado con un ancho Y y un largo X. Por ende los ejes X e Y estan invertidos en referencia a los ejes cartesianos.

El primer jugador compra piezas hasta que se le acaben los recursos depositándolas en un Array y una vez que termina el otro jugador hace lo mismo. Luego el primer jugador pone las piezas en el tablero en un espacio delimitado dependiente del tamaño del tablero.

Al depositar las piezas el primer jugador las va quitando de una copia del Array donde se encuentran las piezas que compró. Una vez que deposita todas las piezas el otro jugador hace lo mismo. Una vez que arranca la partida el jugador puede controlar todas las piezas. Las piezas poseen el numero de jugador al que pertenecen, el match en el que toman lugar, vida y movilidad dependiendo de su clase(guerrero, mago, etc) y un rango de ataque y daño dependiendo del arma que poseen.

Las armas son una Class que tiene parte de la información sobre la pieza que la utiliza y un array de 3 skills para que la pieza use. Además de todo esto tienen otras variables que son los modifiers y attacked y moved. Los modifiers como su nombre implican son modificadores que cambian cosas como la movilidad, la cantidad de daño recibido, y la cantidad de daño dado. Una vez que los modificadores toman efecto toman su forma original. Los skills de las armas tienen cargas. Cuando las cargas se acaban, no se puede volver a usar el skill en toda la partida, salvo que con otro skill se puedan sumar cargas.

Attacked y moved son las variables que determinan lo que la pieza es capaz de hacer en el turno actual. Las piezas arrancan el turno con ambas variables en false (son booleanas) y al moverse cambian Moved a True. Sin embargo al atacar o utilizar un skill, ambas variables se ponen en True lo que significa que una pieza puede moverse y luego atacar pero no atacar y moverse. Las habilidades de la piezas cuentan como ataques en este sentido.

En cualquier momento el jugador puede terminar su turno ya sea porque se quedo sin nada que hacer o porque no quiere hacer ninguna jugada más. Una vez que se pasa el turno todas las piezas vuelven a tener moved y attacked en estado False. El juego se termina una vez que un jugador elimina todas las piezas del contrincante. El juego chequea si un jugador gana cada vez que una acción que pueda llevar a la victoria ocurre. Finalmente al ganar, se crea un cartel de victoria y se reinicia el juego para jugar una nueva partida.

**Decisiones**

Las decisiones más relevantes sobre el diseño de nuestro código que se tomaron fueron las siguientes:

.-Se decidió que las piezas no solo se diferencian por clase sino que también por las armas que poseen. A estas armas se les dio la responsabilidad de tener las habilidades, además del daño y el rango.

.-Se decidió hacer que el juego consista de un tablero, que tiene un matriz de tiles y que tiene a las piezas. Nos pareció que hacer una matriz iba a ser lo mas adecuado, en vez de hacer un mapa, puesto que necesitamos tener toda la información de todas formas, y la matriz es mas organizada y cómoda a la hora de verlo como un tablero de NxM y la única ventaja del mapa venia si se decidía agregar las tiles solo si contenían algo, pero dificultaba después cuando las habilidades generaban el área donde querían dañar, y recorrer la matriz se hacía innecesariamente complicado. Tampoco iba a ser un mapa muy grande ya que las habilidades de las armas están pensadas para tableros más chicos.

.-Poner constantes y agruparlas en archivos separados para poder modificarlas todas juntas más fácilmente lo que facilitaría el balance del juego y el orden del código.

.-Hacer un pattern strategy para los skills, porque cualquier clase podría tener en potencia cualquier habilidad y al haber muchas habilidades usar un pattern strategy permitía reutilizar mucho mas código y hacerlo mucho mas claro.

.-Para los skills se hizo un método separado que representa lo que el skill va a hacer, checkeando asi que el Tile en el cual se trata de lanzar la habilidad es válido. Se hizo esto para poder hacer una herencia de este método ya que la mayoría de los skills tenia validaciones muy parecidas y se podía agrupar en grupos bien definidos (Habilidades que solo afectan a aliados, etc…)

.-Se hizo también un pattern strategy para el Area en el cual los skills van a afectar, debido a que muchos se repiten. Por ejemplo: si se tienen varios ataques que hacen daño en un área de 3x3. Además facilita saber dónde va a afectar esa skill dependiendo a donde se está lanzando lo que facilita pintar en la vista.

.-Originalmente se quería hacer que match lo pudiese acceder cualquiera como si fuese singleton, pero nos dimos cuenta que esto tenía problemas porque no se podía tener varios match y entonces se decidió que las clases que lo necesitaban lo reciban, como las piezas y las skills cuando van a ser usadas.

.-Se pusieron los métodos de atacar y de moverse en las piezas y no en el match, para que la manera de atacar y moverse pueda variar de pieza a pieza. Nos pareció lo más correcto que sea responsabilidad de ella. Ya que es un comportamiento que es propio de la pieza, y si en algún momento se quiere que alguna de las clases que herede de pieza se mueva de alguna manera distinta, basta con Overridear el método.

.-Cada tile tiene una variable que representa si tiene contenida una pieza, pero además para facilidad y comodidad del código y para permitir a la pieza saber donde esta parada, se opto en colocar en la pieza su posición. Posición que se modifica de forma cruzada con la Tile en la cual la pieza está.

.-Se decidió hacer un controller que posea a la vista y al modelo, para separar la vista del modelo, pero no se hizo que el modelo pueda mandarle mensajes de forma directa al controller sino que este lo contiene y trabaja con lo que el modelo devuelve, porque nos pareció mas organizado que el modelo no sepa nada del controller.

.-Se uso en la vista un grid layout porque era lo que mas servía de la manera más cómoda para nuestra vista simplista.

**Aclaraciones**

En caso de que ocurran imprevistos, asegurarse que esten importadas las librerias de java y Junit. De ser posible mantener todas las carpetas: .settings, bin, doc, src, y los archivos .classpath y .project dentro de la misma carpeta.