Controlador

Resumen

La segunda etapa del proyecto del juego `Alpaca Emblem` consiste en programar la funcionalidad de la interacción entre el usuario y el modelo, es decir la conexión lógica entre la vista y el modelo, creando dos entidades relevantes:

- **Tactician**: Es la entidad que representará a los jugadores del juego y que deberá tener conocimiento de las unidades que poseen, el estado de sus unidades, sus caracteristicas y las acciones que puede realizar
- **Controller**: Es la entidad encargada de manejar el estado del juego en todo momento y de interactuar con el jugador del juego. Tiene la capacidad de reiniciar el juego cada vez que se requiera.

Durante el desarrollo del juego se hará uso de patrones de diseño aprendidos durante el curso y que son necesarios para obtener la funcionalidad que es solicitada.

Mejoras respecto a v1.0

Tras la revisión de versión 1.0 se mejoraron ciertos problemas de diseño que existían en la presente versión 2.0.

Lo primero que se modificó fue el uso de interfaces para diferenciar a los items que atacaban con los items que pueden recuperar. La principal desventaja de no ocupar interfaces es hacer un código que no se extensible, por lo que si se desea añadir nuevos items que sirvan para recuperar, para atacar o incluso algún item que sea para otro fin, no existe una diferencia en el formato hecho anteriormente.

Para una mejor visualización de los resultados logrados con este cambio se muestra el diagrama de clases de la versión anterior comparada a la actual versión:

Patterns Design

Para esta tarea se ocuparon principalmente 2 patrones de diseño para lograr obtener los resultados que se deseaban para la creación de las unidades, items, y mapa (con `Factory Pattern`) y poder detectar el estado del juego en cierto instante, como las acciones realizadas por los tacticians, a través de `Observer Pattern`.

En las siguientes secciones se explicará cada uno de los patrones de diseño mencionados y en qué se ocuparon.

Factory Pattern

El Factory Pattern es el encargado de crear objetos sin exponer la lógica de instanciación al cliente, en este caso el usuario del juego. Toda el trabajo de inicializar los objetos queda oculto.

Se utiliza para crear los items, unidades y el mapa del juego.

Estructura Factory en Units y Items

Para las unidades y los items se estructuró una fabrica desde una interfaz `IFactory<Object>` y un `AbstractFactory<Object>` para aprovechar la herencia de métodos y poder realizar llamadas `super()` al constructor del `AbstractFactory<Object>` para la creación de los objetos.

Otra manera de realizar este procedimiento era creando solamente una interfaz con las clases que implementaran esta interfaz para cada tipo de unidad o item, sin embargo esto genera duplicación de código, pero que es posible solucionarlo utilizando el patrón de diseño **Template Method** creando un `abstract class`.

Otra clase que fue implementada fue `FactoryProvider<Object>` utilizando una `class enum Type<Object>`. La finalidad del Provider es poder obtener un tipo de fabrica de manera simple, entregándole como parámetro el tipo de la clase del objeto que se desea crear. Sin embargo, esta clase sólo es testeada para verificar su funcionamiento y no es utilizada en el resto de código, siguiendo los requerimientos de la tarea.

Por último, antes de crear cada objeto de su respectiva fabrica, el usuario debe ser capaz de conocer los parámetros por defecto que van a crear los objetos, por los que se crean métodos `getters` de estos parámetros para conocerlos.

Items

La fabrica de items debe ser capaz de generar un item con los parámetros por defectos definidos en el inicio del proyecto, estos parámetros fueron:

- Name: nombre del objeto en minúsculas

  Ejemplo: Axe
- Power: 10
- Min-Max Range:
  - min: 1 - max: 2 para items de corto alcance
  - min: 2 - max: 3 para item `bow`

Dado a que en esta etapa no se solicita cambiar los parámetros de los items creados no se crean métodos *setters*, y en esta implementación solo es posible revisar los ajustes por defecto.

La idea del diseño se basa en que al momento de seleccionarse una cierta unidad, el controlador debe ser capaz de poder asignarsela a un tactician

Test

Observer Pattern

Se utiliza para generar la interacción entre el controlador y los cambios generados sobre los jugadores de la partida y sus unidades.

En java 12 la utilización de las clases Observer y Observable están deprecadas y no se recomiendo su uso. Por tanto, el diseño utilizado para la implementación de este patrón fue a través del uso de la interfaz `PropertyChangeListener` haciendo el trabajo de la interfaz Observer (el objeto que observa), y de la clase `PropertyChangeSupport` haciendo el trabajo del Observable (el objeto observado).

El patron se ocupa para saber el estado de las unidades de cada jugador, como tambien para conocer el estado del tactician (HAY QUE VERIFICAR QUE CADA TACTICIAN DEBE TENER UNA)

`moveToSelectedUnit()` Se asume que la unidad del jugador actual puede moverse sobre otras unidades (incluso la del rival) [ES ESTO VALIDO???]

Una unidad puede atacar o intercambiar objetos mas de una vez en esta implementacion, pero deberia poder intercambiarlos solamente 1 vez.

La responsabilidad de que en una celda no hayan dos unidades es responsabilidad del modelo al momento de asignar una posicion a una unidad, la posicion no deberia tener una unidad.

Se asume que las unidades deben conocer al tactician al que pertenecen, los items conocen a la unidad que lo posee, por lo que puede conocer al tactician que lo conoce (un item no deberia estar en otra unidad).

Uso en el programa

El Observer Pattern se ocupa principalmente en los cambios del estado de los atributos que posee cada jugador. Estos cambios son:

- Estado del jugador: En primera instancia **se asume** que ningún jugador puede retirarse del juego cuando quiera. Sin embargo, una unidad cambia de estado cuando se le mueren todas sus unidades.
- Cuando se llega a una cantidad maxima de partidas deberia existir una notificacion que se deben retornar a los ganadores.
- Cuando un jugador ocupa a todas sus unidades se debe enviar un mensaje.
- Cuando se seleccionan las unidades se debe "pasar"
- Al momento de mover a las unidades de cada tactician debe existir un observador del cambio, de esta manera es posible asegurarse que cada tactician mueva a su unidad solamente 1 vez

Test

Tactician

Descripcion

Una entidad *Tactician* representa a un jugador que es manejado por el **Controller**. *Tactician* es el encargado de manejar todas las **instrucciones del usuario** y **delegar mensajes a los objetos del modelo** tales como las unidades y los items. Este diseño permite que el usuario **no interactúe** directamente con el modelo del juego.

Para lograr esto tactician también debe **conocer a todas las unidades que posee**, como también tener conocimiento del **mapa del juego**.

Dentro del turno del jugador este puede mover a todas sus unidades, pero **solo una vez**. Esto también aplica con los ataques, una unidad **no puede atacar 2 veces***. Para facilitar la implementación, *tactician* tiene la referencia a la unidad que tiene actualmente seleccionada.

Un jugador debe tener la capacidad de **ver los datos** de sus unidades (HP current y max, items, inventario, poder, etc).

La pieza más importante de un Tactician es su heroe.

Si el héroe de un jugador es derrotado en el turno de cualquier otro, entonces este jugador **pierde la partida y se retira del juego** junto con todas sus unidades. Si el héroe es derrotado en el turno del mismo jugador al que pertenece entonces **se termina su turno antes de ser excluido de la partida**. Un usuario puede tener más de un héroe en juego, en cuyo caso pierde la partida si cualquiera de estos es derrotado.

La unidad que recupera el cleric debe ser del mismo equipo??

- ### Test

Controller

El controller es una pieza fundamental para manejar el estado del juego en cierto instante y para interactuar con el jugador actual, generando acciones entre sus unidades

El controller debe ser capaz de conocer todo respecto al jugador actual o puede saber todo de los demás jugadores? , respecto a esto mismo, el controller puede mover las unidades de un jugador a pesar que no es el jugador actual que deberia jugar?? En mi implementacion asumi que el controller puede seleccionar la unidad que quiera en el mapa, de esta manera puede intercambiar con quien sea, moverse adonde el controller diga. La cosa es que en caso que la unidad seleccionada esté en el inventario de unidades del tactician, entonces se modifica una

Ganar un Juego

El controller es el encargado de revisar cuales son los jugadores que ganaron el juego en ciertas circunstancias. Los ganadores se obtienen **una vez que el juego es finalizado** o que se retiran todos los jugadores.

En detalle, para que un jugador gane:

- **Todos el resto de los jugadores se han retirado del juego:** esto significa que un jugador en una partida actual puede "colocar" me retiro. Esto implica que un tactician cambia su estado a "no seguir", esto lo vamos a definir como un boleeano true si esta activo, false si no (a penas un jugador tenga estado false debe ser eliminado de la partida con sus unidades)
- **Se alcanza una cantidad máxima de turnos** (-1 es indefinido). El ganador en este caso es el que tiene la mayor de unidades restantes. Si dos jugadores tienen la misma cantidad de unidades entonces se declara un empate entre ambos jugadores.

Por tanto si se alcanza una cantidad máxima de turnos entonces el juego debe ser terminado y retornar los ganadores de la partida.

- ### Test

Modo de uso

Supuestos

Los jugadores pueden seleccionar la unidad que quieren

para ver si un jugador puede seguir jugando se crea canPlay(). Un jugador puede seguir jugando si y solo si

Todos sus heros live

Para las unidaddes que tiene cada jugador vamos a considerar los siguientes supuestos:

Casos bordes:

- En cada ronda de juego los jugadores deben usar su turno. AL comenzar la partida se decidirá de forma aleatoria el orden en que juegan los tactician, y al **final** de cada ronda se seleccionará de manera aleatoria un **nuevo orden** de juego de **manera aleatoria**
- Si las unidades pueden tener 0 unidades en un comienzo. En este caso el jugador no puede jugar, esto significaria que tiene que no se puede mover, lo cual quita un poco la logica del programa. Luego tambien sucede que el *tactician* pierde en un caso cuando no tiene mas unidades. Entonces en este caso el jugador que parte sin unidades se ve como que revive. Nah que ver (DEBEN TENER MINIMO 1 UNIDAD
- Cada tactician puede mover a todas sus unidades **solo una vez**, esto implica que una vez que mueva una unidad el proceso de partida continua si hay otra unidad que se pueda mover, es decir, hay otra unidad que este en la misma posicion
- En ningun momento del juego debe haber dos unidades en la misma casilla. Por tanto, si una unidad muere, hay que verificar que la casilla en la que se encuentraba queda vacia para que otra unidad pueda ocupar ese puesto en otro turno, o incluso en el mismo.
- Se asume que el van a existir
- Al seleccionar una unidad esta debe pertenecer al equipo del jugador. No deberia poder seleccionar a otra unidad que no sea de su equipo. MENTIRA, la unidad seleccionada no necesariamente pertenece al jugador actual

Dudas

- Los Tacticians deben tener referencia a todo el mapa y además deben conocer el lugar que se le asigna al comienzo?
- Dado el tamaño del mapa existe una cantidad limitada de jugadores que pueden participar y una cantidad maxima de unidades que pueden tener cada uno para colocarlos en el mapa. Por tanto, se debe considerar esto al iniciarse el juego?
- Asumiendo que tener 0 unidades significa perder, pensé en crear un hero en el constructor de tactician para que tenga sentido que el juego no inicie con todos perdiendo, por ejemplo si el jugador no selecciona a un hero al inicio. Tengo la duda si esto esta bien y si el area que se le asigna al jugador debe ser aleatorio o eso es algo arbitrario?
- Cuando se señala que el controller debe manejar inputs, esto significa que debemos implementar un BufferedReader en Controller?
- En la tarea se debe manejar errores con IOException y todo eso?
- En el enunciado aparece que al jugador se le asigna un área de inicio donde debe situar sus unidades

- Si un metodo retorna un valor cualquiera, es valido que en algun caso retorne nulo? por ejemplo cree un mapa que guarda en las llaves la posicion del hero en la lista de unidades de tactician, y en los valores guarda un boleano que representa el estado del hero (true vivo, false muerto). Esto hace mas facil revisar cuando un jugador perdió o ganó, porque si no habria que revisar cada HP de cada unidad siempre que se termina una batalla.
- Se asume que las unidades parten con un HP, movement??

- Tiene sentido definir como class static a la fabrica de unit e items?
- Una unidad puede atacar e intercambiar solo 1 vez en las siguientes entregas? un Cleric puede recuperar a toda unidad que quiera o tiene que ser de su equipo? Se puede intercambiar con quien sea? https://www.u-cursos.cl/ingenieria/2019/2/CC3002/1/foro/o/23985016
- Como se añade un item a una unidad?
- el controller debe poder mover todo lo que quiera cuando quiera o depende del jugador que esta jugando actualmente?? https://www.u-cursos.cl/ingenieria/2019/2/CC3002/1/foro/o/23994416
- La unidad del jugador actual se puede mover encima de la unidad de otro jugador? o debe ser una restricción del juego?

La victoria depende de las unidades que se tiene

Mientras que las unidades que tenga no sean Hero, pierdo si se mueren todas

-> units.size() == 0

Mientras que una unidad que tenga sea Hero, pierdo si se me muere 1

-> hero.currentHP == 0

Para el que observer funcione en HERO debe ser la sgte secuencia de acciones:

al añadir un hero, a este hero se le añade un observador

el objeto observado es la vida de hero, pero el que añade al observador es un support

cuando la vida del hero cambia, esto es avisado al observador.

El observador conoce:

- La fuente del evento, el cambio, y lo que era antes.

Si la vida del hero llega a 0, entonces directamente el jugador pierde y sale del juego

Para el caso de tener