CC3002 - Metodologías de Diseño y Programación
14 de agosto de 2019
Santiago, Chile

Alpaca Emblem

Tarea 1: Que pase el modelo

Profesor: Alexandre Bergel
Auxiliares: Juan-Pablo Silva
Ignacio Slater
Semestre: Primavera 2019

Resumen

Un cliente quiere hacer un juego de estrategia por turnos con distintos tipos de unidades que tienen fortalezas y debilidades entre ellas. Su cliente intentó implementar el juego por si mismo pero se dio cuenta de que no tenía los conocimientos necesarios para lograrlo.

Para facilitar su trabajo, la implementación se dividirá en 3 entregas. Para esta tarea, su objetivo será implementar los elementos faltantes del modelo del juego. Además debe asegurarse que el código que entregue siga las buenas metodologías de diseño vistas en el curso. Para lograr lo anterior, es libre de modificar tanto como quiera el código recibido.

1. Código Base

El código que se le entrega tiene una implementación de gran parte del modelo del juego. Además, contiene *tests* que prueban su funcionamiento y está documentantado para que su comprensión sea más fácil[1].

El código lo puede encontrar en *GitHub* siguiendo este enlace: `https://github.com/islaterm/cc3002-alpaca-project-template`. El repositorio dado es un *Template* a partir de el cual tendrá que crear su propio repositorio. Para lograr esto, puede ver las instrucciones presentes aquí: `https://help.github.com/en/articles/creating-a-repository-from-a-template`.

Las entidades del juego se presentan a continuación.

1.1. Unidades

Actualmente el juego cuenta con 6 tipos de unidades, las cuales comparten las siguientes características:

Hit points: Es la cantidad de daño que puede recibir la unidad antes de quedar fuera de combate (i.e. la unidad no puede seguir utilizándose y deja el campo de juego). Para esto, se tienen 2 contadores: uno que indica los *hit points* máximos de la unidad, y otro que indica la cantidad actual.

Movement: Representa la cantidad máxima de celdas del mapa que puede desplazarse una unidad. Esto significa que en cada turno una unidad puede ubicarse en cualquier posición del mapa que se encuentre entre 0 y *movement* celdas de distancia desde su sitio actual.

Location: Es la ubicación actual de una unidad en el mapa.

Items: Es una lista con los objetos que porta la unidad. Además, dependiendo del tipo de objeto, algunas unidades podrán equiparse alguno de estos.

[1] No puede asumir que el código que se le entrega está completamente testeado ni que implementa buenas metodologías, juzgar esto es parte de su trabajo.

Exceptuando la *Alpaca*, todas las unidades pueden portar a lo más 3 objetos.

Los tipos de unidades se listan a continuación:

Unidades básicas:

- **Archer:** Sólo pueden equiparse *Bows*.
- **Fighter:** Sólo pueden equiparse *Axes*.
- **Sword Master:** Sólo pueden equiparse *Swords*.

Unidades especiales:

- **Alpaca:** No pueden equiparse ningún tipo de objeto, pero pueden cargar una **cantidad ilimitada** de ellos.
- **Cleric:** Sólo pueden equiparse *Staffs* y **no pueden realizar ataques**.
- **Hero:** Sólo puede equiparse *Spears*. Al ser derrotado, el jugador que perdió esta unidad pierde la partida (esto no está implementado en el código base, ni es parte de esta entrega).

1.2. Objetos

Los objetos son elementos que una unidad puede ocupar sobre otra. Para ocupar un objeto, la unidad primero debe equipársela.

Cada objeto tiene un rango definido en [minRange, maxRange] y sólo pueden utilizarse en unidades que estén dentro de ese rango. El rango de un objeto está acotado inferiormente por 0 y el máximo debe ser estrictamente mayor que el mínimo.

El efecto que genera cada uno de estos al ser utilizado se especifíca en la sección 2.

1.3. Mapa

El mapa se puede pensar como una grilla de dimensiones $n \times n$ en la que cada casilla puede **ser parte del mapa o no**.

Más específicamente, el campo de juego se define como un **grafo en el que cada nodo representa una celda del mapa** y pueden estar o no conectada a otras celdas. Una celda está conectada a otra si están adyacentes (por ejemplo, la celda (0, 0) puede estar conectada con la 0, 1) y se referencian una a otra como vecinos.

Para simplificar, puede asumir que la distancia entre todos los nodos que están directamente conectados tienen distancia 1 entre ellos.

La figura 1 ilustra la estructura del mapa. De este ejemplo, se tiene que la distancia entre las celdas (0, 0) y (0, 1 es 1, mientras que la distancia entre (0, 0) y (2, 0) es 6. Adicionalmente, se tiene que las ubicaciones (1, 0) y (1, 1) no pertenecen al campo de juego, por lo que no se puede llegar ni pasar por ellas.

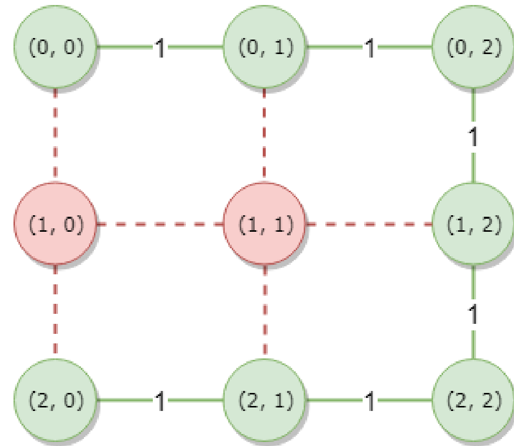

Figura 1: Representación del mapa como un grafo

1.3.1. Location

Son los **nodos que componen el grafo**.

Cada uno tiene una **fila** y **columna** que identifican su ubicación en el mapa y un conjunto de **referencias a sus vecinos directos**. Cada *Location* puede tener una referencia a la **unidad que ocupa la casilla** (en caso de existir alguna).

1.3.2. Field

El mapa del juego. Se representa como un **conjunto de nodos *Location***, para añadir nodos al mapa se utiliza el método `addCells(connectAll: boolean, cells: *Location)`[2]. El primer parámetro indica si los nodos añadidos deben estar conectados a todos sus vecinos, mientras que el segundo es una cantidad indefinida de las ubicaciones que se van a agregar al mapa. Si `connectAll` es `false`, entonces los nodos que se agreguen al grafo se conectarán a sus vecinos de manera aleatoria.

Es importante que el mapa se **mantenga conexo** en todo momento. El método `isConnected` sirve para comprobar la conectividad del grafo.

2. Requisitos

Su cliente le entrega lo que lleva del juego y le solicita que extienda el código entregado para implementar nuevas funcionalidades y terminar de diseñar el modelo. Además le comenta que aprendió a programar siguiendo tutoriales de *YouTube*[3], así que no está seguro del diseño de su solución.

Los detalles específicos de lo que se le pide implementar puede verlos a continuación.

2.1. Combate

Una unidad puede utilizar el objeto que tiene equipado sobre otra siempre y cuando la otra unidad se encuentre dentro del rango definido por el *item*. Cuando esto sucede se entra en un combate.

[2] `cells` puede recibir 0 o más parámetros del tipo *Location*

[3] No tiene una formación profesional en programación.

Cuando se combate, la unidad que lo inició utiliza su objeto sobre la otra, y en caso que utilizar el objeto resulte en un ataque y que la unidad que recibió el ataque pueda atacar, entonces realizará un contrataque. Si en **cualquier momento del combate** una de las unidades participantes es derrotada, el combate finaliza.

Para esto se necesita poder diferenciar entre 2 tipos de objetos:

- Los objetos que **pueden atacar** realizan daño a sus oponentes cuando son utilizados. Además si una unidad ataca a otra que puede atacar, entonces esta última realizará un contrataque (para todo este proceso se debe tomar en cuenta el rango de las armas).
- Existen también objetos que **no realizan ataques**. Éstas no activan un contrataque por parte de la otra unidad, pero pueden ser atacadas por otras, en cuyo caso tampoco responden a este ataque.

Note que si una unidad no tiene equipado un objeto, entonces no puede atacar ni contratacar.

En el código que se le entrega todas las armas realizan ataques, a excepción de los Bastones (*Staff*) que puede sanar a otras unidades. Además, se tienen los Arcos (*Bow*) que no pueden tener un rango menor a 2 (i.e. no pueden atacar ni contratacar a unidades adyacentes). Es importante que se ponga en el caso en que se quisieran agregar más objetos que pertenezcan a cada uno de estos tipos de manera sencilla.

Adicionalmente, algunos objetos son **fuertes contra otros**. Cuando un objeto es fuerte contra otro, su daño aumenta en 1.5 veces, mientras que si es débil contra otro, entonces su daño disminuye en 20 puntos. Las debilidades y fortalezas se muestran en el cuadro 1.

Item	Debil contra	Fuerte contra
Hacha	Espada	Lanza
Espada	Lanza	Hacha
Lanza	Hacha	Espada

Cuadro 1: Debilidades y fortalezas de las armas

2.2. Intercambio

Todas las unidades pueden dar y recibir objetos de otras, siempre y cuando estas estén a distancia 1 entre ellas y que no se supere la cantidad máxima de objetos que puede portar.

2.3. Sorcerer

Se le pedirá además que implemente un nuevo tipo de unidad con su propio tipo de objetos. Un *Sorcerer* es una unidad que puede equiparse libros de mágia y atacar con ellos. Existen 3 tipos de magias, **Luz**, **Oscuridad** y **Ánima**, con sus propias debilidades y fortalezas como se muestran en el cuadro a continuación.

Item	Debil contra	Fuerte contra
Ánima	Oscuridad	Luz
Oscuridad	Luz	Ánima
Luz	Ánima	Oscuridad

Cuadro 2: Debilidades y fortalezas de las magias

Además de la tabla anterior, las mágias son al mismo tiempo fuertes contra todas las armas que no son mágicas y todas las armas que no son mágicas son fuertes contra las mágias (v.g. si se ataca con una mágia de luz a una unidad que tenga espada, entonces realizará 1.5 veces su daño, pero también recibirá 1.5 veces el daño de la espada).

2.4. Requerimientos adicionales

Además de una implementación basada en las buenas prácticas y técnicas de diseño vistas en clases, usted además debe considerar:

- **Cobertura:** Cree los tests unitarios, usando JUnit 5, que sean necesarios para tener al menos un coverage del 90 % de las líneas por paquete. Todos los tests de su proyecto deben estar en el paquete `test`.
- **Javadoc:** Cada interfaz, clase pública y método público deben estar debidamente documentados siguiendo las convenciones de Javadoc[4]. En particular necesita `@author` y una pequeña descripción para su clase e interfaz, y `@param`, `@return` (si aplica) y una descripción para los métodos.
- **Resumen:** Debe entregar un archivo **pdf** que contenga su nombre, rut, usuario de Github, un link al repositorio de su tarea y un diagrama UML que muestre las clases, interfaces y métodos que usted definió en su proyecto. **No debe incluir los tests** en su diagrama.
- **Git:** Debe hacer uso de git para el versionamiento de su proyecto. Luego, esta historia de versionamiento debe ser subida a Github.
- **Readme:** Debe hacer un *readme* especificando los detalles de su implementación, los supuestos que realice y una breve explicación de cómo ejecutar el programa. Adicionalmente se le solicita dar una explicación general de la estructura que decidió utilizar, los patrones de diseño y la razón por la cual los utiliza.

Además debe considerar los requisitos que se especifícan en el resumen del proyecto.

3. Evaluación

- **Código fuente (4.0 puntos):** Este ítem se divide en 2:
  - **Funcionalidad:** Se analizará que su código provea la funcionalidad pedida. Para esto, se exigirá que testee las funcionalidades que implementó[5]. **Si una funcionalidad no se testea, no se podrá comprobar que funciona y, por lo tanto, NO SE ASIGNARÁ PUNTAJE por ella.**
  - **Diseño:** Se analizará que su código provea la funcionalidad implementada utilizando un buen diseño.
- **Coverage (1.0 puntos):** Sus casos de prueba deben crear diversos escenarios y contar con un coverage de las líneas de al menos 90 % por paquete. No está de más decir que sus tests deben testear algo (es decir, no ser tests vacios o sin asserts).
- **Javadoc (0.5 puntos):** Cada clase, interfaz y método público debe ser debidamente documentado. Se descontará por cada falta.
- **Resumen (0.5 puntos):** El resumen mencionado en la sección anterior. Se evaluará que haya incluido el diagrama UML de su proyecto. **En caso de no enviarse el resumen con el link a su repositorio su tarea no será corregida**[6].

[4] `http://www.oracle.com/technetwork/articles/java/index-137868.html`

[5] Se le recomienda fuertemente que piense en cuales son los casos de borde de su implementación y de las fallas de las que podría aprovecharse un usuario malicioso.

[6] porque no tenemos su código.