Alpaca Emblem

Tarea 2: Controlando el juego

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

Resumen

A diferencia de la entrega anterior, en esta usted deberá implementar casi completamente el código a partir de su tarea anterior.

El siguiente paso en la implementación será crear un **controlador**. Un controlador es una interfaz[1] que comunica el modelo del juego con un *posible usuario*, por lo que debe manejar todos los *inputs* de este. Luego, el controlador **envía mensajes** a los objetos del modelo de acuerdo al *input* recibido. En consecuencia, el usuario no interactúa directamente con el modelo[2].

Se le entrega un archivo base para implementar el controlador. Usted deberá completar este archivo con los métodos necesarios para cumplir con los requisitos. Puede[3] modificar el código entregado, pero es sumamente importante que **no elimine ni cambie la firma** de ningún método, ya que se utilizarán para testear algunas de las funcionalidades que se le solicitan.

1. Requisitos

Esta entrega parte de la base de que su anterior tarea se implementó correctamente, por lo que, en caso de tener alguno, se le solicita que arregle los errores cometidos.

Para manejar el juego se requiere que implemente 2 elementos más:

- El primero de estos representará a los **jugadores**. En el contexto del juego, se referirá a los jugadores como *tacticians*[4].
- El segundo elemento a implementar es el **controlador**. Esta entidad se encargará de manejar el estado del juego en todo momento y de interactuar con la vista.

A continuación se desarrollan los requisitos específicos para cada entidad.

1.1. Tactician

Como se mencionó anteriormente, la entidad *Tactician* representará a un jugador.

La responsabilidad de esta entidad será la de manejar todas las instrucciones del usuario y delegar mensajes a los objetos del modelo. Esto resultará en que el usuario no interactúe directamente con el modelo del juego.

Un jugador debería ser capaz de realizar todas las acciones requeridas en la tarea 1, por lo que deberá implementar métodos para invocar a todas esas funcionalidades en esta clase. Además, debe asegurarse de que también pueda realizar las acciones presentes en el código base.

[1] No confundir con las interfaces de *Java* (!)
[2] Más adelante verá que el usuario tampoco se comunica directamente con el controlador.
[3] Debe 👀
[4] De aquí en adelante se utilizarán los términos usuario, *Tactician* y jugador indistintamente

Un *Tactician* debe ser capaz de conocer todas las unidades que posee, y tener conocimiento del mapa del juego. Dentro de su turno un jugador puede mover a todas sus unidades, pero **una sola vez**. Para facilitar la implementación el jugador debe mantener una referencia a la unidad actualmente seleccionada.

Adicionalmente, un usuario debe tener la capacidad de ver los datos de sus unidades, tales como puntos de vida actuales y máximos, nombre, inventario, poder, etc.

1.2. Controlador

Se le entrega una plantilla para implementar el controlador. Tenga en cuenta que esta implementación está incompleta y tendrá que modificarla para que maneje todos los posibles *inputs* de un usuario.

No debe modificar la firma de los métodos presentes en el código entregado ya que será utilizado para evaluar el correcto funcionamiento de su código, en caso contrario no se obtendrá puntaje en esta sección. Aparte de lo anterior, es libre de modificar esta clase tanto como desee.

El controlador debe mantener el estado del juego en todo momento, considerando los siguientes campos:

- **Tacticians**: El controlador debe monitorear a todos los jugadores de la partida. Una partida puede tener una cantidad arbitraria de jugadores.
- **Mapa del juego**: Debe haber una referencia al mapa del juego.
- **Turno actual**: El *tactician* que está jugando actualmente.

Cualquier cambio en el estado del juego debe ser notificado al controlador para que este decida qué hacer en cada situación.

En cualquier momento de su turno un jugador puede decidir no realizar más acciones, en cuyo caso termina su turno y se pasa al turno del jugador siguiente.

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.

Una ronda de juego se definirá como un ciclo en que todos los jugadores usen su turno. Al comenzar una partida se decidirá de forma aleatoria el orden en que jugarán los usuarios, y al final de cada ronda se seleccionará un nuevo orden de juego de manera aleatoria, con la restricción de que un jugador no puede tener dos turnos seguidos.

Para comenzar una partida, el controlador debe ser capaz de:

- Crear a los jugadores. Para simplificar esto, implemente el controlador de tal forma que los jugadores tengan nombres "Player 1", "Player 2", etc. en el orden que se vayan creando).
- Iniciar y asignar a sus unidades.
- Crear un mapa aleatorio

A cada jugador se le asigna un área de inicio donde debe situar sus unidades (el orden en que los jugadores ubican sus unidades será el mismo que el orden de turnos de la primera ronda). La idea aquí es que cada jugador pueda seleccionar las unidades que desea utilizar en la partida y ubicarlas en el mapa, pero esta mecánica no es un requisito para esta entrega. Basta con que haya una manera sencilla de crear unidades y que se puedan asociar a algún jugador.

En ningún momento del juego puede haber dos unidades en la misma casilla, pero si una unidad es derrotada entonces se retira de su celda.

Existirán dos maneras de ganar el juego:

1. Todo el resto de los jugadores se han retirado del juego.
2. Se alcanza una cantidad máxima de turnos (considere que -1 significa que se jugará sin limite de puntos). El ganador en estos casos se define como el jugador con mayor número de unidades restantes. En caso de que dos o más jugadores tengan la misma cantidad de unidades la partida se declara empatada.

Además de la base para implementar el controlador, se incluyen los *tests* mínimos por los que tendrá que pasar su trabajo para que su tarea se considere correcta. Note que faltan los *tests* para comprobar la creación y asignación de unidades, la creación y asignacion de objetos, los combates y las condiciones de victoria. Se le solicita que implemente los *tests* necesarios para probar dichas funcionalidades.

1.3. Requerimientos adicionales

Además de una implementación basada en las buenas prácticas y técnicas de diseño vistas en clases, usted 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[5]. 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 especifican en el resumen del proyecto.

2. Evaluación

- **Código fuente (4.0 puntos):** Este ítem se divide en 2:
  - **Funcionalidad (1.5 puntos):** Se analizará que su código provea la funcionalidad pedida. Para esto, se exigirá que testee las funcionalidades que implementó[6]. **Si una funcionalidad no se testea, no se podrá comprobar que funciona y, por lo tanto, NO SE ASIGNARÁ PUNTAJE por ella.**
  - **Diseño (2.5 puntos):** 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).

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

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

- **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**.[7]

[7] Porque no tenemos su código.