Tarea 1 - Era de los Imperios

Profesor: Alexandre Bergel
Auxiliar: Juan-Pablo Silva

En esta tarea se le pedirá que implemente un modelo de peleas inspirado en el clásico juego de estatégia *Age of Empires*. Su aplicación debe estar debidamente testeada y documentada, y debe utilizar buenas prácticas de diseño. Especificaciones serán dadas a continuación en este documento.

Requisitos

Tendremos 2 tipos de entidades, entidades que pueden recibir ataques (`Attackable`) y entidades que pueden atacar (`Attacker`). Las entidades `Attacker` pueden atacar y, a su vez, recibir ataques por lo que también son `Attackable`. Dentro de estos 2 tipos hay otra clasificación, esta corresponde a las construcciones (`Building`) y unidades (`Unit`). Sabemos que todas las unidades pueden atacar, pero esto no es el caso para las construcciones. De las construcciones, la única que puede atacar son los castillos (`Castle`).

Las unidades (`Unit`) se dividen en los siguientes tipos: `InfantryUnit`, `ArcherUnit`, `CavalryUnit`, `SiegeUnit`, `Monk` y `Villager`. Todas las unidades pueden atacan y recibir daño. Las construcciones (`Building`) se dividen en `Barracks` y `Castle`. A pesar de ser ambas construcciones, a diferencia de los castillos, las barracas no pueden atacar. Para ser más explícito: las `Barracks` solo pueden recibir daño y los `Castle` pueden atacar y recibir ataques.

Age of Empires es un juego de estratégia, pero en esta tarea nos centraremos solamente en las interacciones entre entidades al momento de ~~humillar a tu oponente~~ atacar. Dentro del juego, distintas entidades tienen ventajas sobre otras, esto responde al patrón de *rock-paper-scissors*, en el que infantería le gana a caballería, caballería le gana a los arqueros, y los arqueros le ganan a la infantería. Por su puesto, el juego es mucho más complejo que esto, pero lo simplificaremos de esta forma para poder abordarlo en la tarea.

Además de esta relación de quién le gana a quién, agregamos que las armas de asedio, como las catapultas, son efectivas contra casi todo, destruyendo todo a su paso salvo a la caballería, que dado son rápidos, logran escapar a tiempo. Los monjes al atacar, en vez de causar daño, recuperan un porcentaje de sus puntos de ataque a la entidad "atacada", pero esto solo se cumple contra las unidades, no recupera *Hit Points* de las construcciones, y a demás mueren al instante al ser atacados por cualquier otra entidad que no sea un aldeano. Para recuperar los *Hit Points* de las construcciones están los aldeanos. Estos son débiles contra prácticamente todo salvo otros aldeanos, pero son capaces de reparar las construcciones y las armas de asedio. Los castillos al ser de piedra ~~se prenden en fuego cuando les disparan flechas~~ son resistentes a espadas y flechas, pero débiles contra armas de asedio. Las barracas son de madera y por lo tanto más frágiles que los castillos,

pero aún así difíciles de destruir con espadas y flechas e igual de fáciles de destruir con armas de asedio. Detalles respecto a cómo interactúa cada una de las distintas entidades se pueden ver en la tabla 1.

Las reglas de este juego (no el original) son las siguientes:

- Una vez los *Hit Points* de una entidad lleguen a 0, esta no podrá seguir atacando.
- Los *Hit Points* de una entidad no pueden caer bajo 0. Es decir, deben ser siempre positivos (asumamos al 0 como positivo).
- Las construcciones tienen un máximo de *Hit Points*, no importa cuánto se reparen, no pueden subir de este máximo.
- Las unidades tienen la particularidad de que cuando se recuperan, pueden llegar a un número de *Hit Points* mayor al inicial que tenían, hasta un máximo del doble de puntos. Es decir, si una unidad partió con 100 `HP` y la atacaron dejandola en 50, un monje podría recuperarla hasta un máximo de 200 `HP`, ya que su moral será mucho mayor que antes.
- No se puede revivir. Una vez una unidad muere o una construcción es destruida (sus *Hit Points* llegan a 0), esta no puede ser ni recuperada por un monje ni por un aldeano, significando que una vez sus *Hit Points* llegan a 0, la entidad queda inutilizable (puede modelar esto como "hacer nada" al pedirle algo).

La tabla 1 se lee de la siguiente manera: las entidades que están en la columna "Ataca" al atacar a las entidades que están en la fila "Recibe", golpean su número de puntos de ataque multiplicado por el factor indicado en la tabla, o visto de otra manera, las entidades en la fila "Recibe" al ser atacados por las entidades de la columna "Ataca", reciben de daño los puntos de ataque de quien los golpeó multiplicado por el factor de la tabla correspondiente.
Ejemplo: `CavalryUnit` golpea el doble de sus puntos de ataque a `ArcherUnit`, pero no tiene daño extra al golpear a otro `CavalryUnit`.

Usted está en total libertad sobre cómo implementar todas estas entidades, pero fíjese en las definiciones dadas anteriormente y utilicelas para modelar sus clases para que se le sea más fácil. Las entidades `Attackable` tienen métodos para retornar el valor actual de los *Hit Points*, las entidades `Attacker` además de este método, tienen un método para atacar `attack` y otro que retorna los puntos de ataque de la entidad. Los puntos de ataque y *Hit Points* de cada entidad pueden ser recibidos en un constructor o *hardcodeados* por ustedes en el constructor sin argumentos. Queda a su criterio definir cómo se contabiliza el daño dentro de cada entidad. No está de más aclarar que una entidad muere cuando sus puntos de vida llegan a 0 o menos y **no puede volver a atacar**. Usted debe tener un método que verifique que efectivamente la entidad sigue viva (puede llamarlo, por ejemplo, *isAlive*).

Ataca/Recibe	Infantry	Archer	Cavalry	Siege	Villager	Monk	Castle	Barracks
Infantry	1.0	1.2	1.2	1.2	1.5	X	0.3	0.7
Archer	1.2	1.2	1.0	0.8	1.5	X	0.1	0.7
Cavalry	1.0	1.5	1.0	1.2	1.5	X	0.3	0.7
Siege	1.5	1.5	1.0	1.5	1.5	X	2.0	2.0
Villager	0.8	1.0	0.5	0.5 ↑	1.0	–	0.3 ↑	0.7 ↑
Monk	0.5 ↑	0.5 ↑	0.5 ↑	–	0.5 ↑	0.5 ↑	–	–
Castle	1.2	1.2	1.2	0.5	1.2	X	0.1	0.7
Barracks	/	/	/	/	/	/	/	/

Table 1: Matriz de interacciones para las entidades del programa. Note la diferencia entre – y /. – es no ataca o no hace daño, / es no aplica o no *puede* atacar. $N \uparrow$ se refiere a que aumenta los puntos de vida en vez de disminuirlos. X se refiere a que sus puntos de vida se hacen 0 instantáneamente al ser atacado.

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 4, 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[1]. 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.

Evaluación

- **Código fuente (4.0 puntos):** Este ítem se divide en 2:
  - **Funcionalidad:** Se analizará que su código provea la funcionalidad pedida.

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

  - **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**[2].

Entrega

Suba el código de su tarea a Github[3]. Su repositorio de Github debe ser **privado**, y debe llamarse **cc3002-tarea1**, lo cual significa que si su usuario de Github es *juanpablos*, el link a su repositorio será `https://github.com/juanpablos/cc3002-tarea1`. Además, deberán invitar a la cuenta del equipo docente, *CC3002EquipoDocente*, como colaboradores de su repositorio para que podamos acceder a él. Para hacer esto entran a su repositorio, luego a *Settings*, *Collaborators*, ingresan su contraseña y finalmente escriben *CC3002EquipoDocente* en el campo para buscar por usuario, seleccionan *Add collaborator* y nos habrán invitado exitosamente. Las invitaciones se aceptan al momento de corregir su tarea, por lo que no se asuste si aparece como solicitud pendiente.

Usted debe subir a U-Cursos **solamente el resumen**, con su nombre, rut, usuario de Github, link a su repositorio y diagrama UML. El código de su tarea será bajado de Github directamente. El plazo de entrega es hasta el 25 de octubre a las 23:59 hrs. Se verificará que efectivamente el último commit se haya hecho antes de la fecha límite. **No se aceptarán peticiones de extensión de plazo**.

Recomendaciones

No estamos interesados en un programa que solamente funcione. Este curso contempla el diseño de su solución y la aplicación de buenas prácticas de programación que ha aprendido en el curso. No se conforme con el primer diseño que se le venga a la mente, intente ver si puede mejorarlo y hacerlo más extensible.

[2] porque no tenemos su código.
[3] `https://github.com/`

No comience su tarea a último momento. Esto es lo que se dice en todos los cursos, pero es particularmente importante/cierto en este. Si usted hace la tarea a último minuto lo más seguro es que no tenga tiempo para reflexionar sobre su diseño, y termine entregando un diseño deficiente o sin usar lo enseñado en el curso.

Haga la documentación de su programa en inglés (no es necesario). La documentación de casi cualquier programa open-source se encuentra en este idioma. Considere esta oportunidad para practicar su inglés.

Les pedimos encarecidamente que las consultas referentes a la tarea las hagan por el **foro de U-Cursos**, en la categoría "Consultas Tarea: Tarea 1". En caso de no obtener respuesta en un tiempo razonable, pueden hacernos llegar un correo al auxiliar o ayudantes.

¡Éxito!