Informe de documentacion

Model Alpaca Emblem

Profesor: Alexandre Bergel
Fecha: 1 de diciembre de 2019
Link de Trabajo

Autor: Sebastián Sepúlveda A
User GitHub: sesepulveda17
RUT: 19.640.031-1

Resumen

En el presente informe, se presentará un resumen con diagramas *UML* (Unified Modeling Language) de la estructura del código para el controlador del juego *"Alpaca Emblem"*. La estructura del código fue proporcionada a través del siguiente Template [1], donde se añade el paquete `controller` el cual posee métodos incompletos que tuvieron que ser completados durante el proceso del trabajo. También se añaden `Test` para `controller` los cuales son necesarios para implementar de mejor manera los métodos. Para esta etapa se solicitó además implementar un nuevo elemento llamado `Tactician` el cual es el encargado de manejar los cambios generados en cada jugador del juego. Por último, para la correcta comunicación entre los elementos del programa se implementa el patrón `Observer` con el paquete `PropertyChangeListener` y `PropertySupport`. Mientras que para la creación de elementos se ocupa el patrón `Factory`, lo cual genera cambios en la estructura del modelo anteriormente hecho.

En los siguientes diagramas no se incluyen los métodos de cada clase debido a la gran cantidad de métodos en cada una de ellas, lo que dificulta su visualización. Sin embargo se pueden encontrar los archivos UML en los siguientes links: UML_Items, UML_map, UML_units, UML_Controller

Diagrama UML Items

Ver Figura 1. Cambios respecto al *Template*.

- Se añade un nuevo `package FactoryItem` el cual contiene las clases necesarias para la creación de cada item. El tipo `IFactoryItem` permite crear una instancia de cualquier fábrica en este paquete.
- Se añade un nuevo tipo de item: `ItemNull`. Permite no ocupar la instancia del objeto `null` implementado en Java.
- Se crean dos tipos de Item: `IAttack`, aquellos Items que pueden atacar pero no sanar y `IHeap`, aquellos items que pueden recuperar pero no atacar
- Reutilización de código: Se crean clases abstractas para impedir la duplicación de código y reutilizarlo en las distintas clases que heredan el mismo comportamiento del clase abstrata.

[1] Cuerpo Docente CC3002, Segundo Semestre 2019, Universidad de Chile

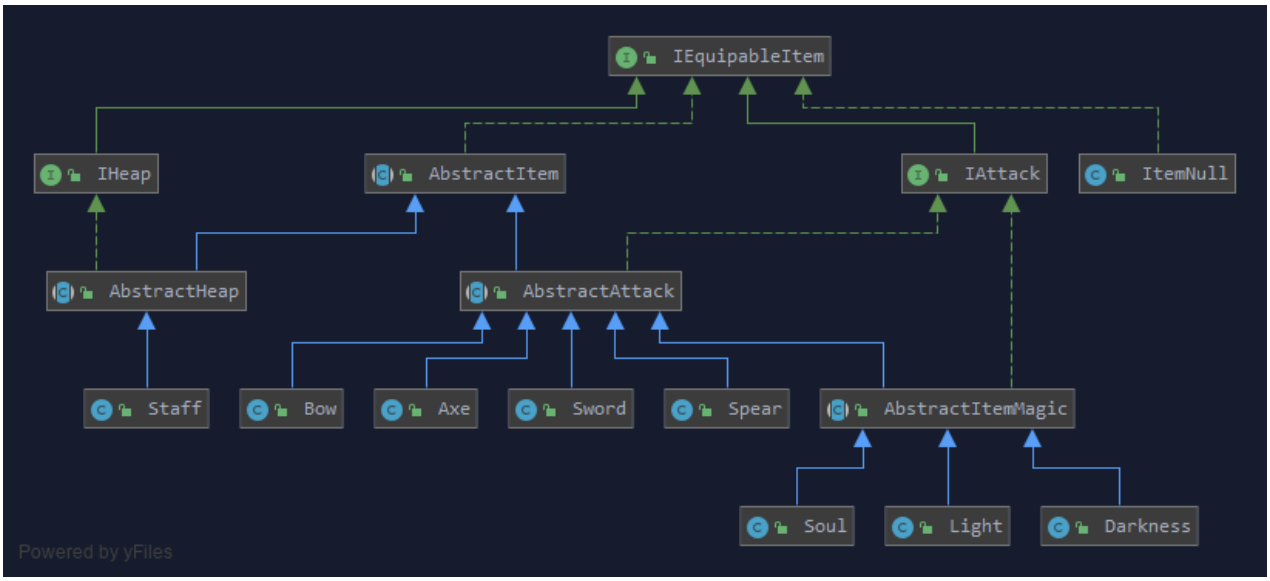

Figura 1: Diagrama UML items

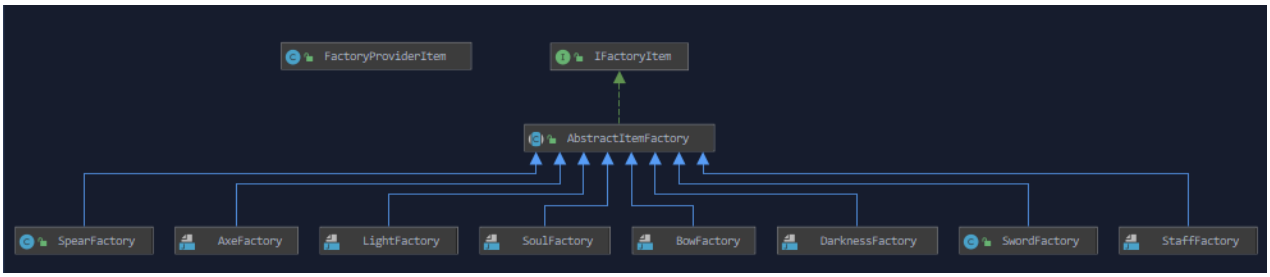

Figura 2: Diagrama UML FactoryItem

Diagrama UML Unit

Ver Figura 3. Cambios respecto al *Template*.

- Se añade un nuevo `package FactoryUnit` el cual contrine las clases necesarias para la creación de cada item. El tipo `IFactoryUnit` permite crear una isntancia de cualquier fábrica en este paquete.
- Nuevos `Handlers`, basandose en el patrón `Observer` se crean estas clases necesarias para detectar los cambios generados en las unidades al añadirse un item en su inventario y para detectar el cambio del HP al recibir un ataque o una sanación.
- Nuevos tipos de unidades y caracterización de comportamiento: se añaden `NormalUnit`, `SpecialUnit`. Como se conoce desde la etapa anterior, las unidades normales no hacen perder al jugador de inmediato, mientras que las unidades especiales hacen perder al jugador al momento de ser derrotadas.
- Se crean clases abstractas `AbstractUnitCombative`, `AbstractUnitCOmbative` para diferenciar el comportamiento entre las unidades que sanan y las unidades que pueden atacar y de esta manera evitar una duplicación de código en futuras modificaciones y con las clases que ya estaban creadas.
- Se añade un nuevo tipo de unit: `UnitNull`. Permite no ocupar la instancia del objeto `null` implementado en Java.

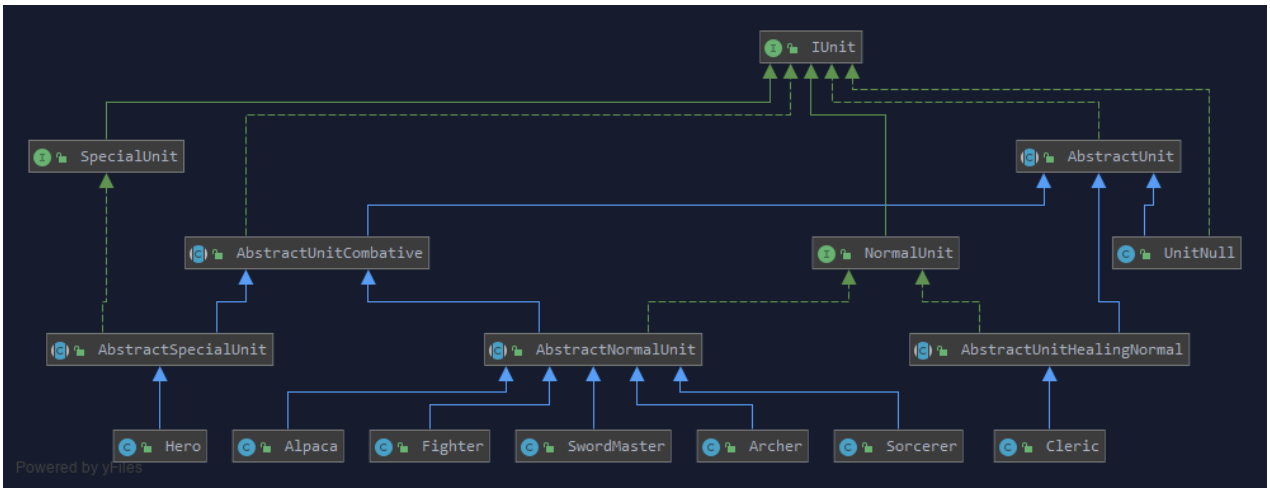

Figura 3: Diagrama UML unit

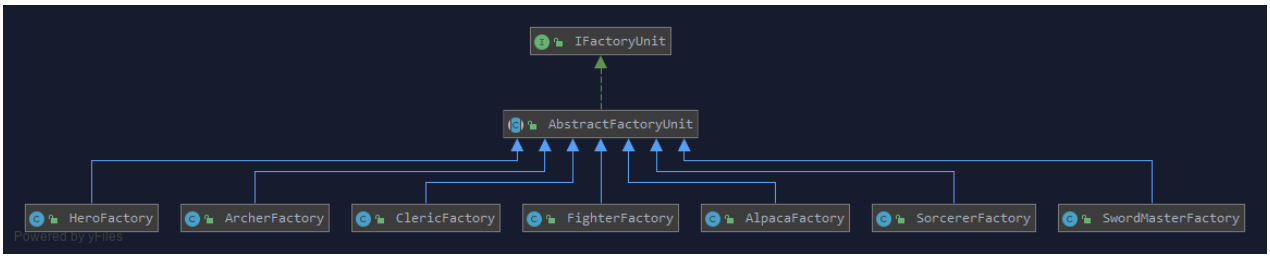

Figura 4: Diagrama UML FactoryUnit

Diagrama UML Map

Se añade `FactoryMap` para la creación simple de un mapa de `Field` con tamaño $n \times n$. **Ver Figura 5**

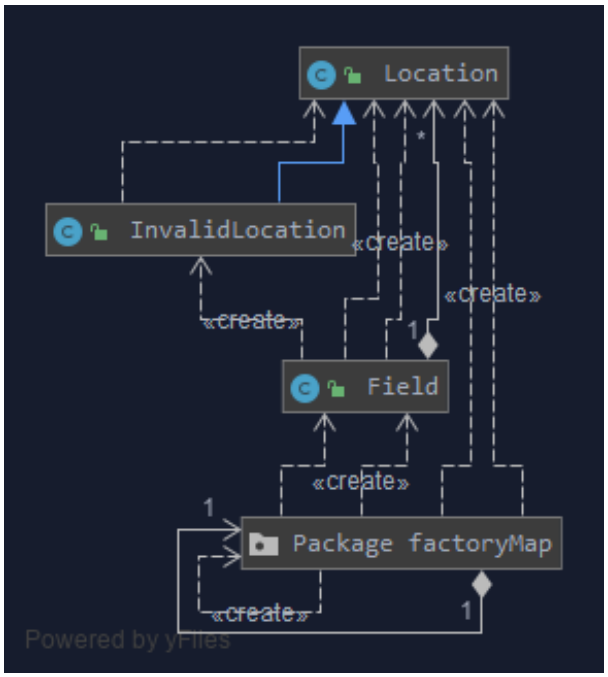

Figura 5: Diagrama UML map

Diagrama UML Controller

Ver Figura 6. Cambios respecto al *Template*.

- Se completa cada método que se solicitaba implementar para el manejo de los `Tactician`, sus unidades, los items, y la creación de los elementos que permitarán el inicio del juego.
- Nuevo elemento del juego `Tactician` encargado en manejar a los jugadores que comienzan a jugar. Por medio de `Tactician` se puede conocer los distintos elementos que posee un Tactician, como también las características de cada unidad.
- Se crean distintos `Handlers` para generar acciones frente a los cambios que se presenten en los `Tactician` respecto a sus unidades, activando métodos presentes en `Controller`. De esta manera es posible eliminar jugadores o realizar cambios respecto a cada cambio que se genere sobre `Tactician`.

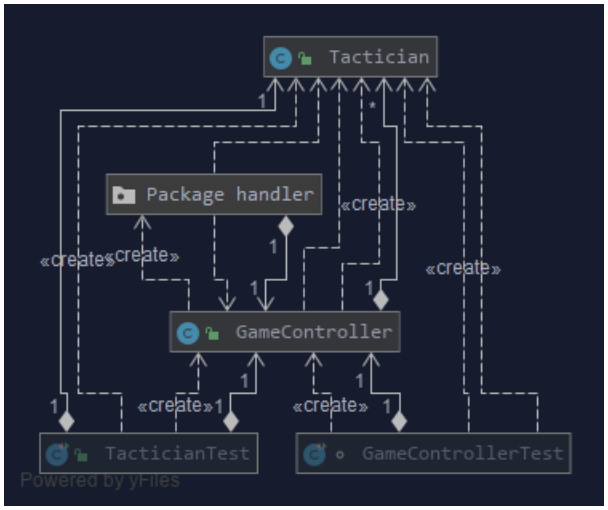

Figura 6: Diagrama UML map