Proyecto 2
Spring Hero

Valerie Michell Hernández Fernández
valeriehernandez@estudiantec.cr
Mariana Navarro Jiménez
mariana12@estudiantec.cr

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación

I Semestre

Viernes 3 de junio de 2022

Resumen

Mediante el presente documento se pretende ofrecer una síntesis de la funcionalidad del proyecto ***Spring Hero***. Este proyecto consiste en desarrollar un juego básico en el que se controla a un personaje capaz de eliminar enemigos y salvar aliados. Para implementar dichas mecánicas se deben de utilizar tres patrones de diseño: el patrón de diseño Observer, Factory y Model-View-Controller.

Palabras clave: patrones de diseño, videojuegos de mazmorras, videojuego, MVC, Factory, Observer, Java Swing

Abstract

This document is intended to provide a summary of the functionality of the project tittled ***Spring Hero***. This project consists of developing a basic game in which a character is capable of eliminating enemies and saving allies. To implement these mechanics, three design patterns must be used: the Observer, Factory and Model-View- Controller.

Keywords: design patterns, roguelike, game, MVC, Factory, Observer, Java Swing

---

Índice

1. Descripción general

Este proyecto consiste en desarrollar un juego básico en el que se controla a un personaje capaz de eliminar enemigos y salvar aliados. Para implementar dichas mecánicas se utilizaron tres patrones de diseño: el Observer, el Factory y el Model-View-Controller.

2. Arquitectura

2.1. Requerimientos

- Java versión 8
- Java SE Development Kit versión 15

2.2. Clases

2.2.1. Diagrama

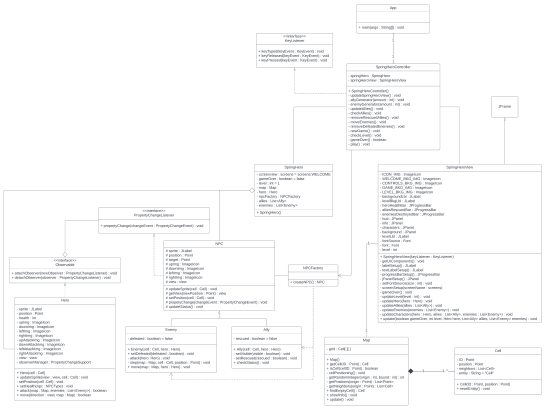

Figura 1. Diagrama de clases UML del juego Spring Hero.

2.2.2. Descripción

App

Clase pública ubicada en el paquete *springhero*, encargada de inicializar el controlador del modelo ***SpringHero*** con su método público *main*.

Constants

Interfaz pública ubicada en el paquete *springhero models main*, la cual contiene variables de tipo primitivo y de tipo compuesto. Así cómo lo indica el nombre de la interfaz, las variables son constantes utilizadas en la mayoría de las clases implementadas en el proyecto.

Figura 2. Resultado gráfico de la ejecución del método *main* de la clase ***App***.

SpringHero

Clase pública ubicada en el paquete *springhero models main*, implementa la interfaz ***Constants***. Esta clase es el modelo en el MVC de la clase ***SpringHero***. Por lo tanto, contiene los atributos necesarios para ejecutar el juego Spring Hero, entre los cuales se puede mencionar las instancias de la clases ***Hero***, las lista de tipo ***Ally*** y la lista de tipo ***Enemy***.

SpringHeroView

Clase pública ubicada en el paquete *springhero views main* desarrollada cómo la vista en el MVC de la clase ***SpringHero***, la cual hereda de la clase ***JFrame*** de ***Java Swing*** e implementa la interfaz ***Constants***.

Figura 3. Resultado gráfico de configuración de GUI en pantalla de control.

Fue desarrollada como la encargada de la visualización de los componentes de la aplicación. Por lo anterior, esta posee desde métodos genéricos para la creación de etiquetas, paneles y barras de progreso hasta métodos encargados de la actualización de la

posición de las etiquetas de las entidades de tipo ***Hero*** y ***NPC*** enviadas por el controlador y recibidas mediante su método update.

Figura 4. Resultado gráfico de configuración de GUI en pantalla de juego.

La relación de herencia entre la clase ***SpringHeroView*** y la clase ***JFrame*** surge por la necesidad de implementar una interfaz gráfica al juego. De esta forma, la disposición de todos los métodos de la clase facilita la implementación de los gráficos de la aplicación.

SpringHeroController

Clase pública ubicada en el paquete *springhero controllers main* desarrollada cómo el controlador en el MVC de la clase ***SpringHero***, esta clase implementa las interfaces ***Constants*** y ***KeyListener*** de Java AWT. Esta clase implementa los métodos que cubren la lógica del algoritmo del juego, la cuál progresa mediante la información que recibe del usuario mediante el ***KeyListener***, actualizando constantemente la vista ***SpringHeroView*** y la información del modelo ***SpringHero***. Cabe recalcar que mediante el método keyPressed el controlador es capaz de obtener la información del usuario y reenvía esta información a la vista para actualizar la posición de sus componentes y configura la información en el modelo.

Map

Clase pública ubicada en el paquete *springhero models main* implementa la interfaz ***Constants***, la clase ***Map*** fue desarrollada con el objetivo de cubrir los aspectos lógicos de la matriz del tablero. Implementando un arreglo 2D compuesto por entidades de tipo ***Cell***, ***Map*** cubre el posicionamiento y cálculo de posibles posiciones cercanas a una ***Cell*** especificada. De esta forma, ***Map*** podría verse como un simple tablero y las entidades ***Hero*** y ***NPC*** como las fichas que se ubican sobre el tablero. Además, en su constructor hace uso de su método *cellPositioning*, donde instancia cada ***Cell*** según su posición. En este caso por ser una matriz el cálculo de la posición de cada ***Cell*** dependerá del tamaño definido en la interfaz ***Constants***.

Cell

Clase pública ubicada en el paquete *springhero models main* implementa la interfaz ***Constants***. El objetivo principal de esta clase es tener una entidad con información en cada celda del grid del ***Map***. No obstante, en la parte lógica es primordial pues el arreglo 2D de la clase anterior está compuesto por esta entidad y durante la ejecución del juego es necesario saber cuales casillas del tablero están vacías, de esta forma se posicionan las entidades de tipo ***Hero*** y ***NPC*** y esta clase posee el atributo Cell.entity que determina esa disponibilidad.

Hero

Clase pública ubicada en el paquete *springhero models hero*, encargada de generar el objeto de tipo ***Hero***. Esta clase implementa los métodos para movilizar, atacar y aumentar o disminuir la vida del héroe del juego según su entorno. Por otra parte, cabe recalcar que esta clase implementa dos interfaces, ***Constants*** y ***Observable*** cómo parte del patrón de diseño Observer, el propósito de la anterior es el de notificar a la clase ***NPC*** sobre el movimiento y posición del héroe de esta forma los NPC pueden moverse acorde a la posición del héroe.

Figura 5. Sprite de la clase ***Hero***.

NPC

Clase pública ubicada en el paquete *springhero models npc* implementa las interfaces ***Constants*** y ***PropertyChangeListener*** de Java Beans. Ofrece una entidad con los atributos necesarios de un NPC

en el juego, entre lo más revelantes se puede mencionar su etiqueta para mostrar en la vista del MVC, la posición actual y su objetivo, siendo este último la posición del héroe en el juego. Es importante mencionar que la implementación de la interfaz de Java Beans surge de la necesidad de implementar el patrón de diseño Observer, mencionado en la clase ***Hero*** y explicado más adelante. Dónde esta le proporciona al NPC recibir constantemente la posición actual del héroe, esto con el fin de actualizar su posición en el caso de los enemigos o actualizar su visibilidad en el caso de los aliados. Además, todo NPC posee cuatro vistas, la actual de estos dependerá de la posición de su objetivo.

Ally

Clase pública ubicada en el paquete *springhero models npc*, subclase de la clase ***NPC***. ***Ally*** ofrece una entidad que abarca una ***Cell*** en el grid del juego.

Figura 6. Sprite de la clase ***Ally***.

Cada uno de estos posee dos estados: rescatado y visible. Un aliado sólo puede ser rescatado por el héroe, en caso de que el héroe se encuentre se coloque en la misma celda del aliado este será rescatado y sumará a la vida del héroe. La visibilidad del aliado dependerá de la posición del héroe, el héroe debe encontrarse en un rango de distancia de tres celdas para poder visualizar al aliado, en caso contrario el aliado permanecerá no visible al jugador.

Enemy

Clase pública ubicada en el paquete *springhero models npc*, subclase de la clase ***NPC*** con variedad de sprites. ***Enemy*** proporciona una entidad capaz de infligir daño al ***Hero***, aumentan durante los niveles de una partida y sólo pueden destruidos por el héroe. Estos se mueven justo después de que el usuario mueva al héroe mediante el teclado. Su dirección dependerá de la posición actual del héroe, en el proceso se actualizará su etiqueta con el fin de dar una mejor experiencia gráfica al usuario. Por lo tanto, en caso

Figura 7. Sprite de la clase ***Enemy***.

de que el ***Hero*** haya sido destruido por los ***Enemy*** la partida terminará y mostrará un mensaje de juego terminado.

NPCFactory

Clase pública ubicada en el paquete *springhero models npc* que implementa la interfaz ***Constants***. El objetivo de esta es ser la clase principal del patrón de diseño Factory, de esta forma el controlador en lugar de generar una instancia directa de las clases ***Ally*** y ***Enemy*** cada que necesite una de estas, el controlador únicamente debe llamar a su instancia de la clase y enviarle el tipo de ***NPC*** que desea y la celda dónde desea posicionarlo.

Observable

Interfaz pública ubicada en el paquete *springhero models observer*. El propósito de esta interfaz es el de crear un mecanismo que le permite notificar a diversas clases o objetos si objeto al que se esta observando fue modificado. Cabe recalcar que esta clase posee dos métodos: atachObserver y detachObserver. Dichos métodos son utilizados para vincular al objeto observador a un objeto que esta siendo observado.

3. Patrones de Diseño

3.1. Model-View-Controller (MVC)

El patrón de diseño conocido cómo MVC fue implementado en este proyecto con el propósito de manejar la parte lógica y la parte gráfica de la aplicación de manera individual. Dónde el modelo lleva el nombre de ***SpringHero***, la vista el nombre de ***SpringHeroView*** y el controlador el nombre de ***SpringHeroController***. Dónde el modelo únicamente posee las instancias de los atributos del juego y sus respectivos métodos de get y set, la vista contiene toda referencia gráfica del juego y el controlador

contiene los métodos lógicos del algoritmo del juego. De esta forma el flujo de comunicación entre la vista y el modelo es únicamente regulada por el controlador. Dónde este, ejecuta la simulación del juego y actualiza la información almacenada en el modelo y seguidamente manda la información gráfica a la vista para que esta la muestre.

Entre las ventajas del patrón MVC se encuentra principalmente la organización y manejo de errores, pues conforme aumenta el tamaño y la complejidad del algoritmo una sola clase puede alcanzar una gran cantidad de líneas de código. Sin embargo, entre las mayores desventajas se encuentra lo largo del flujo de comunicación e inclusive lo tedioso que se puede volver actualizar constantemente ambas clases (modelo y vista) en lugar de hacerlo directamente en un mismo método.

3.2. Factory

Cómo se menciona en la descripción de la clase ***NPCFactory***, este patrón de diseño ofrece facilidad al algoritmo de instanciar objetos. Permitiendo inclusive crear nuevas variaciones de una entidad sin necesidad de modificar el código principal. Reduciendo el tiempo de creación de manera que aumenta considerablemente la consistencia del algoritmo. Sin embargo, la complejidad del patrón puede aumentar junto a la cantidad de subclases que posea la clase base.

3.3. Observer

Para implementar el patrón Observer se utilizó la interfaz ***Observable***. Esta interfaz contiene dos métodos attachObserver y detachObserver. Estos métodos se utilizan para vincular al objeto Observador, que en este caso son los NPC a un objeto que esta siendo observado el héroe. El uso este patrón de diseño proporciona diversas ventajas pues se logra mantener actualizado a múltiples objetos sobre el comportamiento o estado del objeto observado en todo momento y además permite mejorar la jerarquía en el código al proveer métodos específicamente para observar y notificar. No obstante, si el observado tiene muchos observadores este proceso puede tardar en notificarles, provocando que el rendimiento del programa disminuya.

4. Recursos

Durante el desarrollo del proyecto se utilizaron recursos externos tanto para el diseño de la interfaz gráfica como para la implementación de los algoritmos desarrollados en la solución final.

4.1. Interfaz Gráfica

- `Javax Swing - Oracle [manual]`
- `Swing Tutorial - Youtube [video]`

4.2. Algoritmos

- `Multidimensional Arrays in Java - Geeks For Geeks [article]`

4.3. Patrón de Diseño

- `Model View Controller - MDN [article]`
- `Observer - Refactoring Guru [Web Page]`