FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 2
Primer cuatrimestre de 2023

Alumnos:	Número de padrón:	Email:
Peralta, Federico Manuel	101947	fperalta@fi.uba.ar
Zimbimbakis, Francisco Manuel	103295	fzimbimbakis@fi.uba.ar
Garcia Sanchez, Julian	104590	jgarcias@fi.uba.ar
Lozano, Martina Victoria	106267	mlozano@fi.uba.ar
Natale, Nicolas Marcelo	108590	nnatale@fi.uba.ar

Índice

1. Introducción

Se busca desarrollar una aplicación llamada *'AlgoDefense'* de manera grupal aplicando todos los conceptos vistos en el curso, utilizando un lenguaje de tipado estático (Java) con un diseño del modelo orientado a objetos y trabajando con las técnicas de *TDD* e *Integración Continua*.

2. Supuestos

Durante el desarrollo del proyecto se hicieron uso de una serie de supuestos (*no explicitos en la consiga del mismo*) necesarios que se dictan debajo.

- Los *enemigos* se generan en la primer pasarela del camino (considerada la largada).
- El *camino* (por el cual se mueven los enemigos) dado en formato *JSON* siempre debe ir de arriba a abajo e izquierda a derecha.
- La *Trampa de Arena* al atacar (reduccion de velocidad en 1) a un enemigo con velocidad 1, lo deja quieto durante la vida util de la defensa.
- Las defensas atacan al primer enemigo en rango que encuentran.
- El *movimiento diagonal de la lechuza* utiliza el algoritmo *Bresenham* (utilizado para calcular aproximaciones a la hipotenusa de un triangulo rectangulo) para determinar el siguiente movimiento de dicho enemigo con cada nueva ubicacion, lo cual puede generar una pseudo diagonal.
- Los enemigos al llegar a la meta atacan una vez y desaparecen.
- El enemigo *Lechuza* al morir no da creditos al jugador.
- En una *parcela* puede haber *defensas* y *enemigos* a la vez.
- No se pueden ubicar *defensas* sobre una *parcela* con *enemigos*.

3. Modelo de dominio

El proyecto se desarrollo en torno a la clase *Juego*, la cual recibe por inyeccion de dependencias un *Jugador*, el *mapa*, los *turnos* y un *lector* para leer estos 3 ultimos dados en formato *JSON*.

El *Jugador* creado tiene un nombre con un minimo de 6 y un maximo de 10 caracteres recibido por la interfaz grafica, un valor de 20 puntos de vida y 100 creditos para gastar.

En el transcurso del juego el usuario puede agregar *defensas* en las distintas *parcelas* disponibles, se iran generando *enemigos* en el inicio del *camino* que avanzaran dentro de sus respectivas capacidades para eventualmente morir a causa de las *defensas* o lograr dañar al *jugador* al llegar a la *meta*.

Toda interaccion/comportamiento no permitido por el *Juego*/*Interfaz Grafica*, levantara un tipo de excepcion (de las cuales hablaremos mas adelante) para comunicarse con el usuario.

4. Diagramas de clase

A continuacion se presentan los diagramas de clases del modelo del proyecto correspondiente.

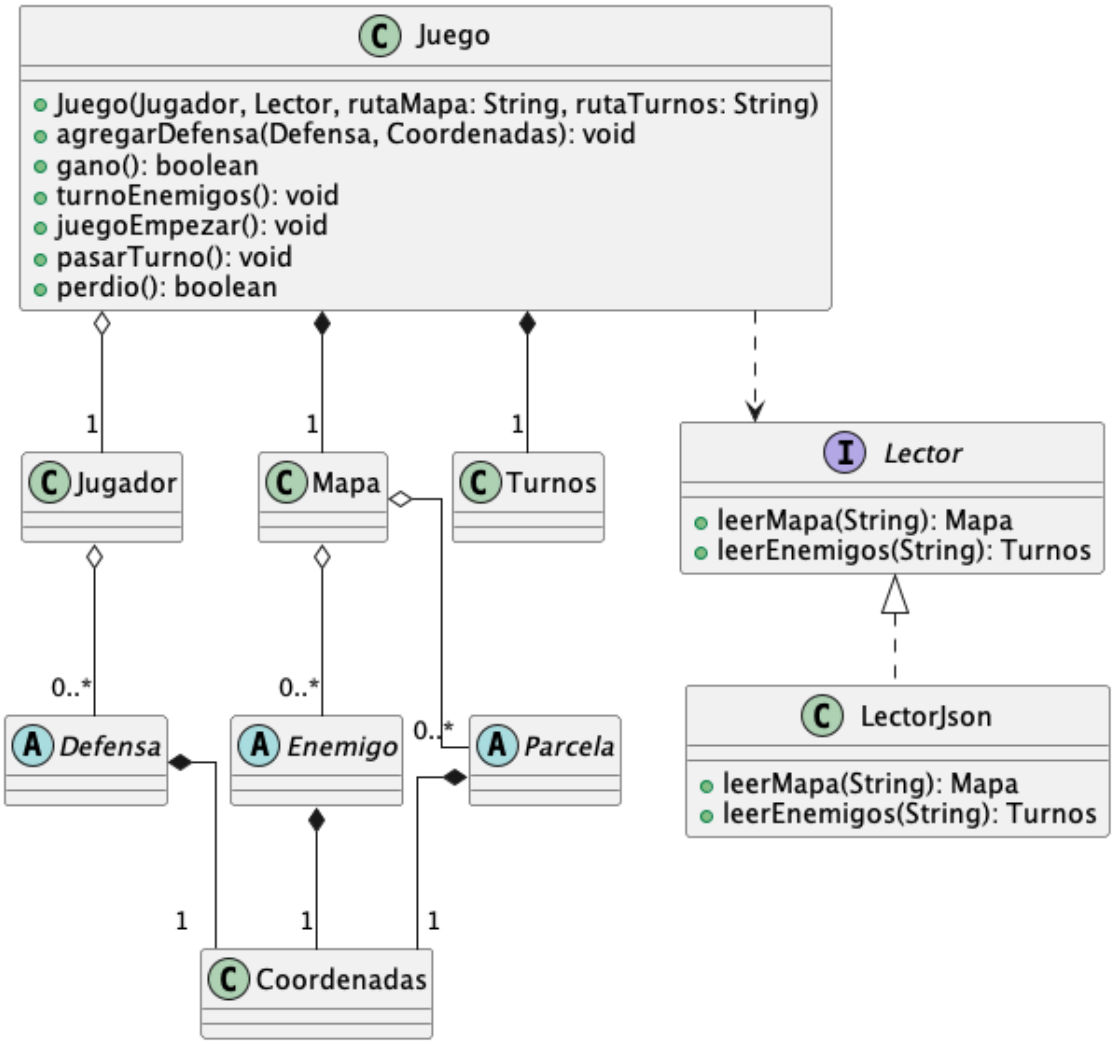

Figura 1: Diagrama del Juego.

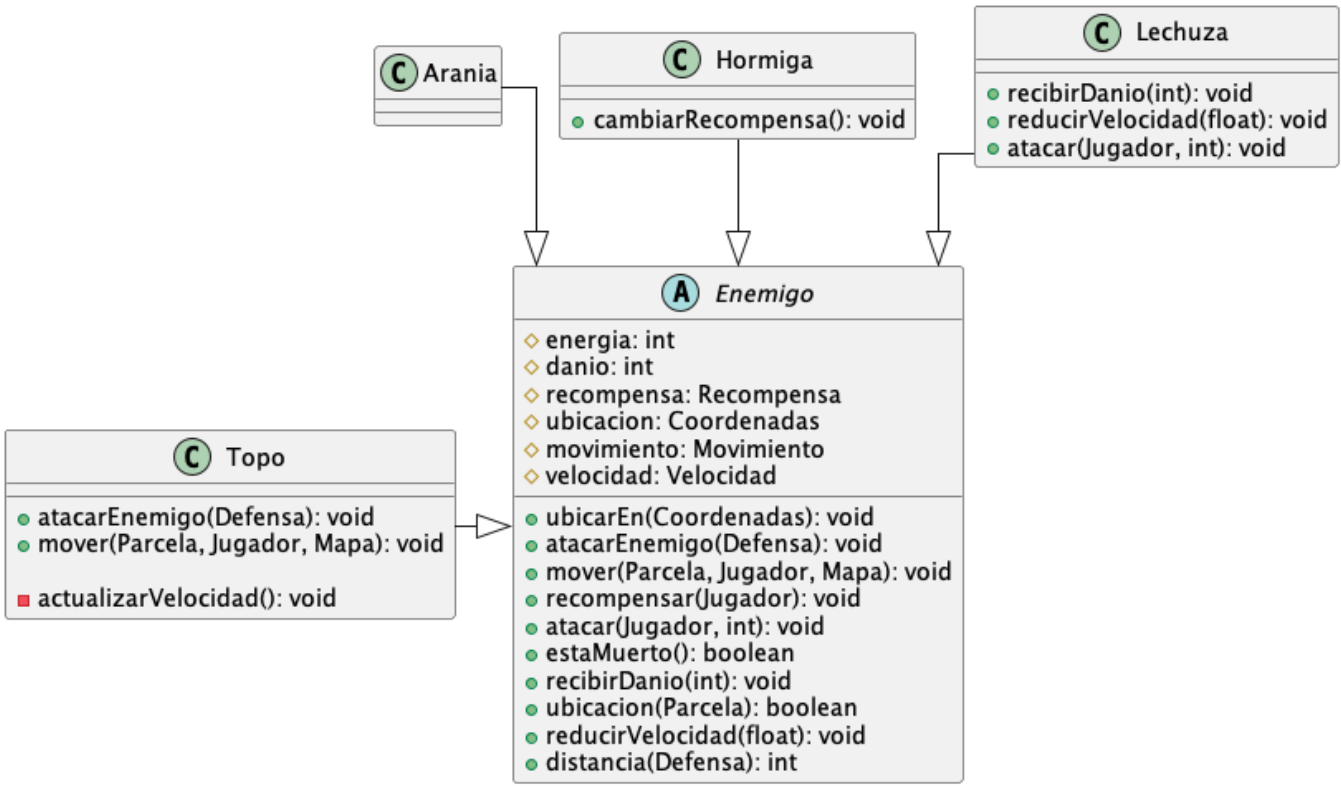

Figura 2: Diagrama de familia de enemigos.

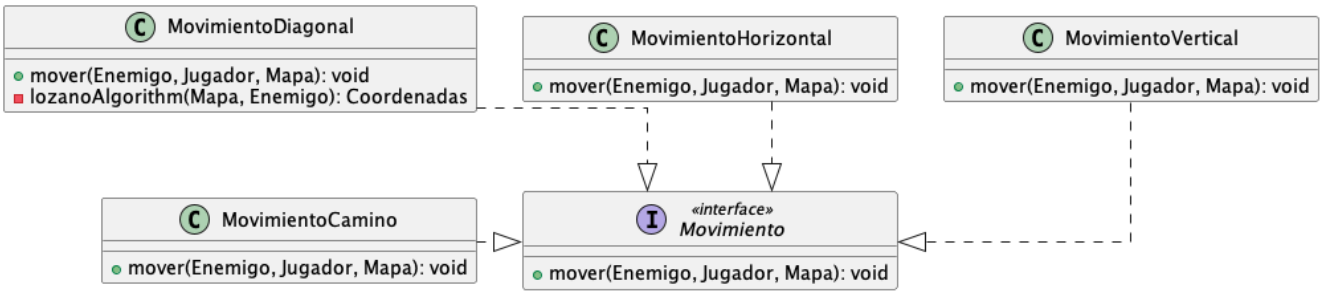

Figura 3: Diagrama movimiento de los enemigos

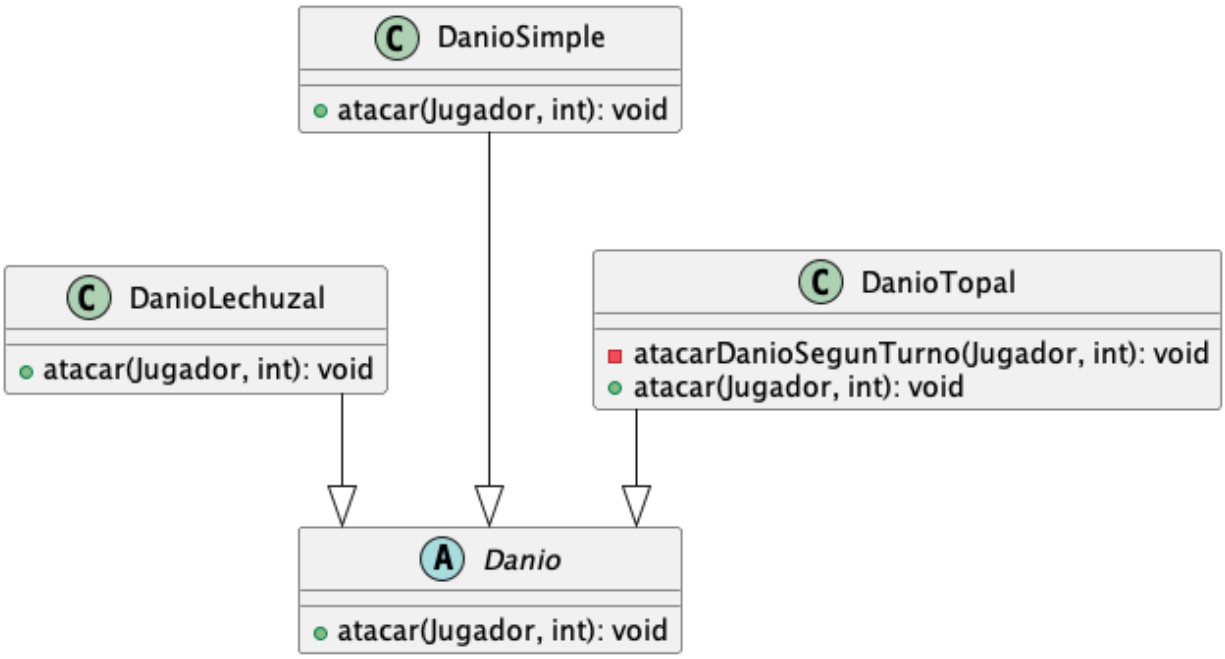

Figura 4: Diagrama ataques de los enemigos

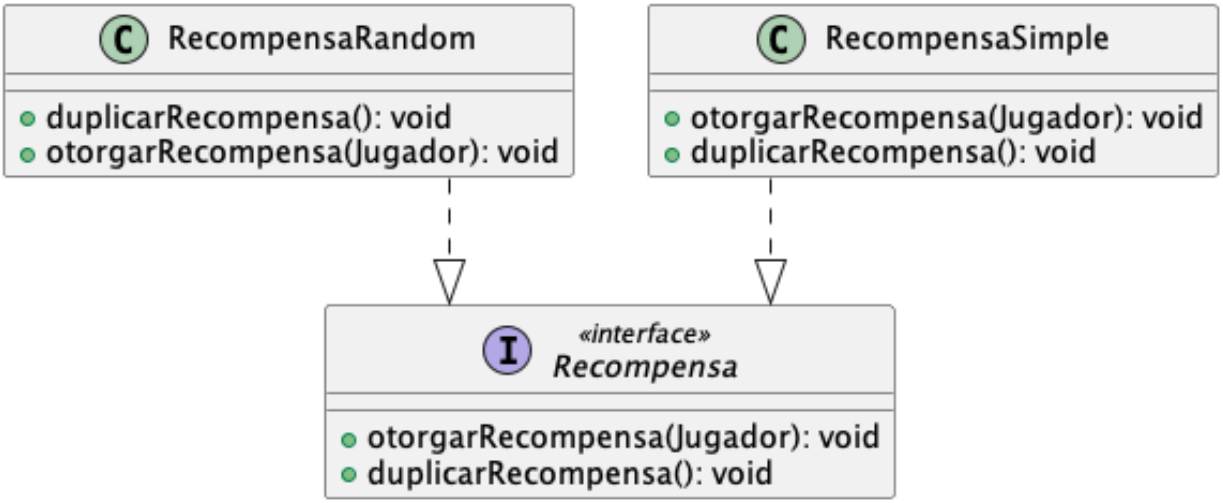

Figura 5: Diagrama recompensas que dan los enemigos

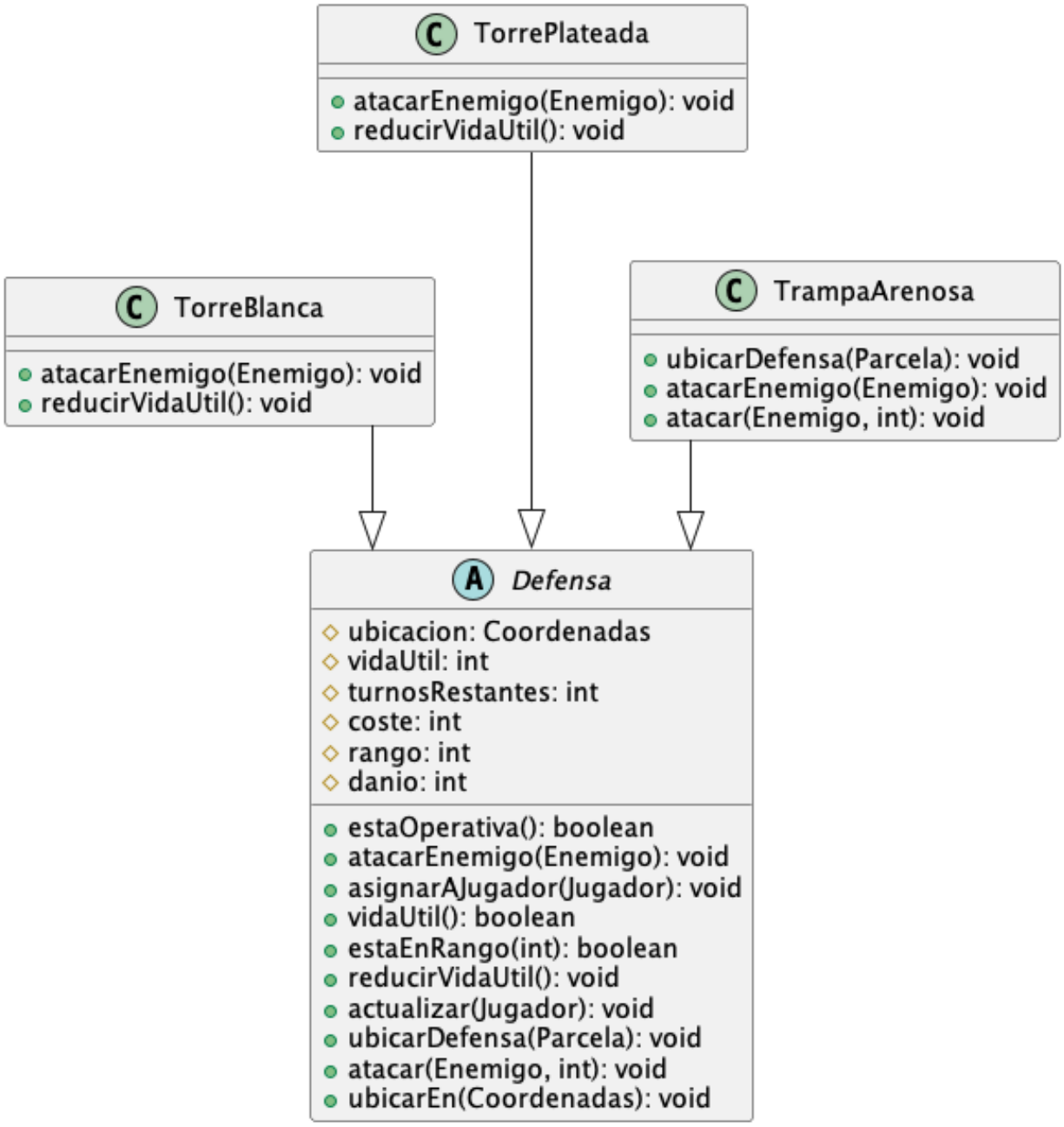

Figura 6: Diagrama familia de defensas

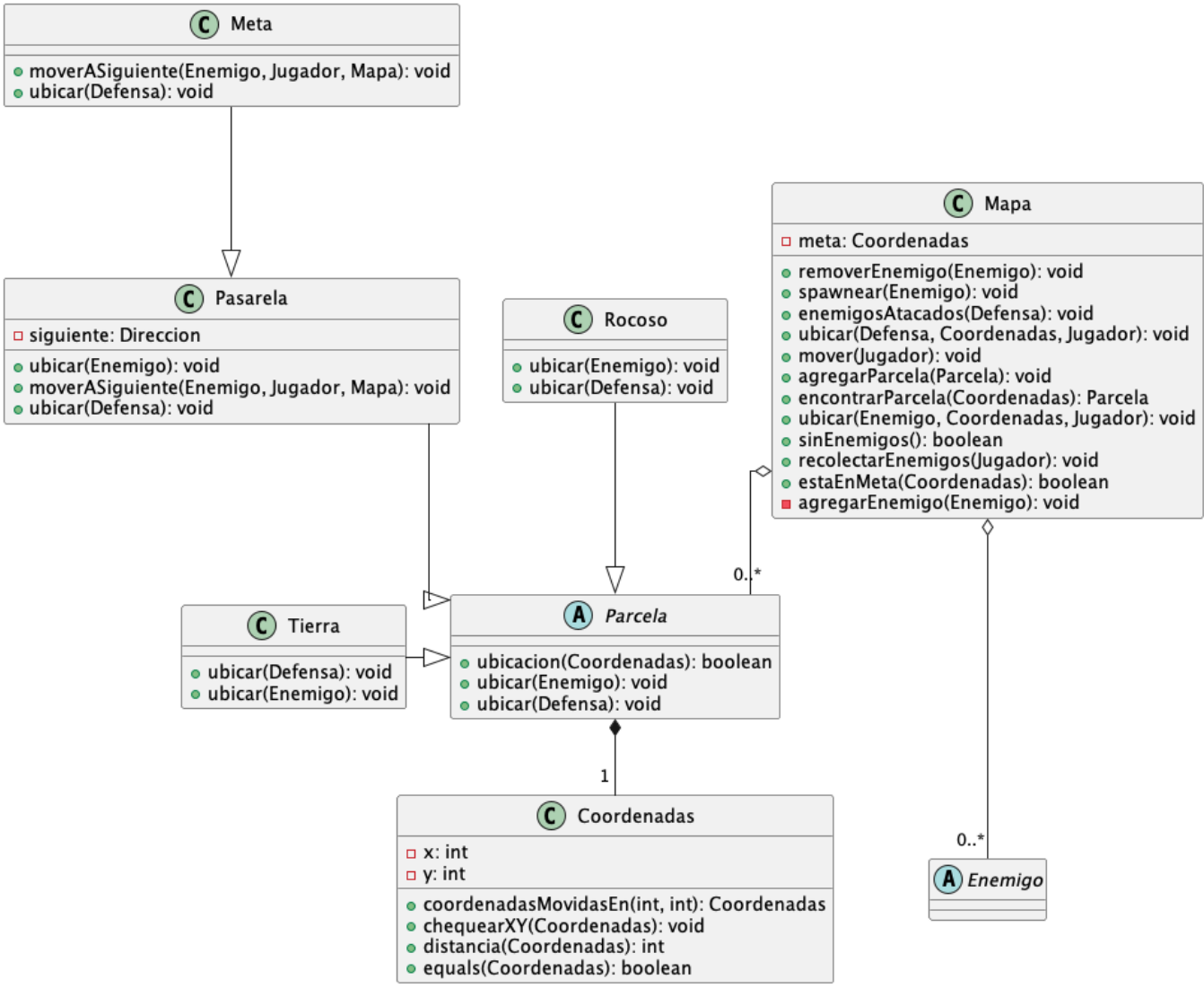

Figura 7: Diagrama mapa del juego

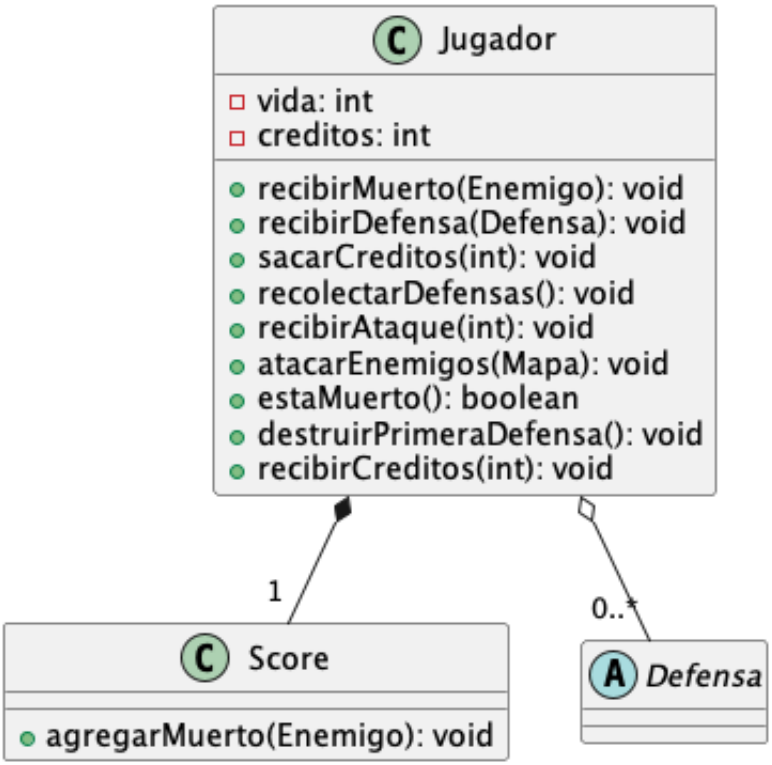

Figura 8: Diagrama jugador

Figura 9: Diagrama abstract factory

5. Diagramas de paquetes

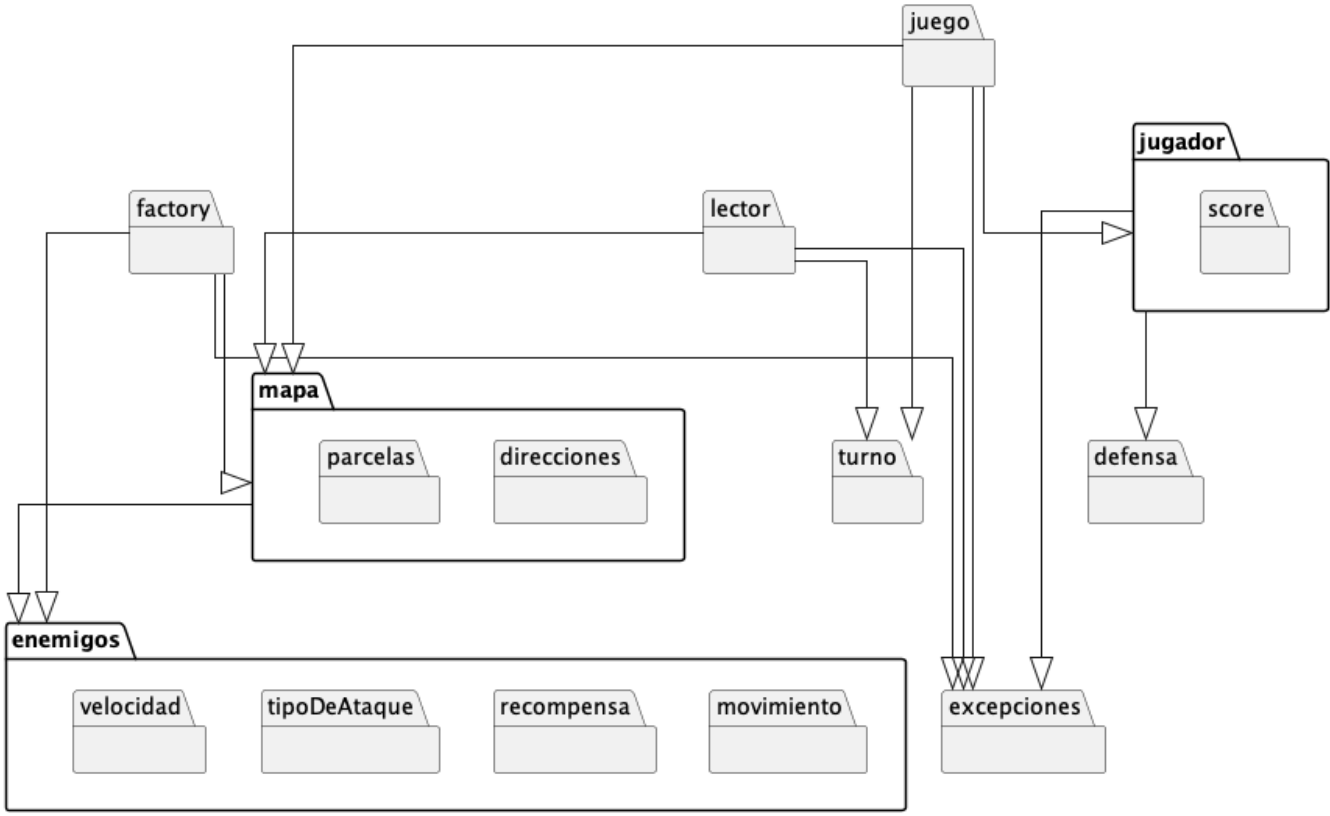

Figura 10: Diagrama general de todos los paquetes

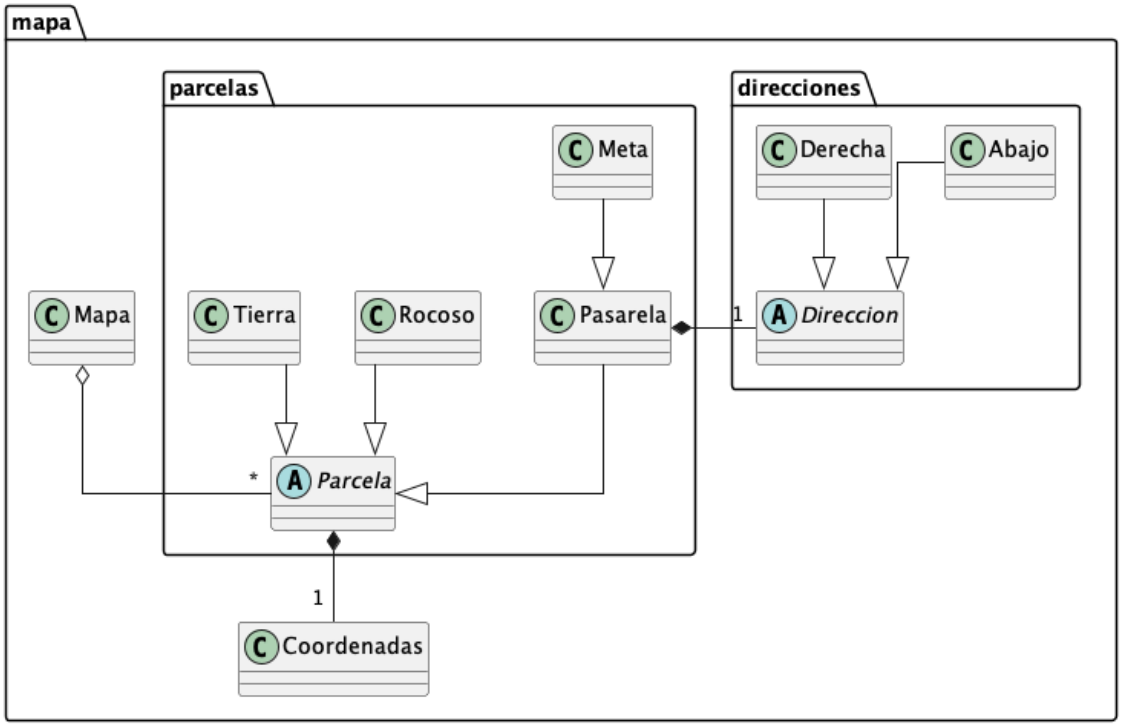

Figura 11: Diagrama interno de todos los paquetes de Mapa

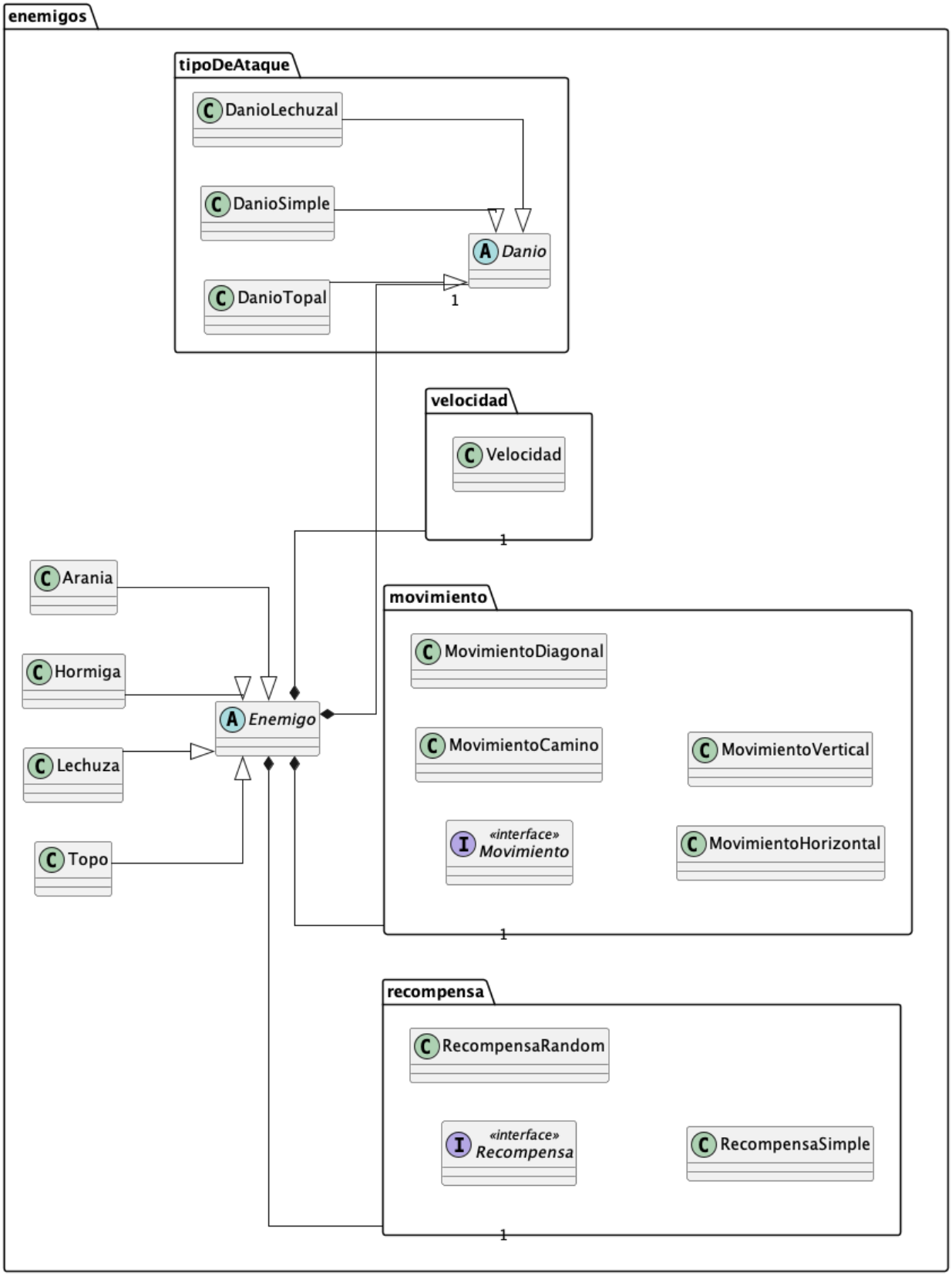

Figura 12: Diagrama interno de todos los paquetes de Enemigos

6. Detalles de implementación

6.1. Presentacion de clases y el Juego

Para modelar el proyecto nos basamos en los pilares de la programacion orienteda a objetos, haciendo uso de distintos tipos de patrones de diseño y buenas practicas. La clase *Juego* es la clase que interactua con el usuario. Con ella se agregan las *defensas*, se pasan los *turnos* e inyecta las dependencias en todos los movimientos del programa. Recibe por parametros al *Jugador*, y una serie de urls para el correcto armado del *Mapa* y las oleadas de *Enemigos*.

6.2. Defensas

Las Defensas son una familia de clases, todas heredan de la clase madre las cualidades que tiene una defensa concreta, tales como tener una cantidad de vida y costo de construccion, poder atacar a un enemigo, tener una ubicacion y un tiempo de construccion. Posteriormente, cada una de sus subclases implementa o setea estas de forma correspondiente a sus especificaciones.

6.3. Enemigos

Siguiendo la misma idea de las *Defensas*, todos los *Enemigos* tienen una clase abstracta de la cual heredan cualidades generales, que posteriormente se implementan especificamente en cada subclase. Estas cualidades son: tener *Vida*, un *tipo de Ataque* hacia el *Jugador*, una *Velocidad*, una *Ubicacion* y un *Movimiento*. Esta ultima es la mas llamativa, ya que no todos los *Enemigos* tienen el mismo tipo de *Movimiento*, para este tipo de comportamientose utilizo el patron de diseño *Strategy*, con el cual podemos settear en tiempo de ejecucion los cambios que puede llegar tener un *Enemigo* durante el transcurso del juego, un ejemplo seria la *Lechuza* que al recibir suficiente daño para dejarla en 50 % de vida su *Movimiento* sobre el *Mapa* se vuelve diagonal.

6.4. Jugador

El Jugador es una clase concreta diseñada para representar al usuario. Este almacena las *Defensas* del mismo, recibe daño de los *Enemigos* al llegar a la *Meta*, recibe y utiliza los creditos que tiene disponibles. Este contiene una instancia de una clase *Score*, la cual verifica y (si es necesario) settea las recompensas de los enemigos muertos.

6.5. Coordenadas y Direccion

Toda clase con una ubicacion hace uso de una *Coordenada*. Esta clase *Coordenada* da la posibilidad de comparar posiciones entre distintas clases. Las subclases de *Direccion* permiten hacer cambios en la ubicacion de los *Enemigos* de forma mas abstracta.

6.6. Parcelas

Las *Parcelas* modelan los distintos tipos de terrenos. Cada una es responsable de saber si pueden ubicar una *Defensa* o *Enemigo* en particular. Estas hacen uso del patron Double Dispatch para saber que tipo especifico de *Defensa* o *Enemigo* es el que se le quiere ubicar. Cabe mencionar que la subclase *Pasarela* utiliza una *direccion* para saber su siguiente *Pasarela* en el *Camino*.

6.7. Mapa

El *Mapa* tiene la responsabilidad de contener todas las *Parcelas* del *Juego* y los *Enemigos* dispersos en este. Es el que conoce los limites del plano y ordena las ubicaciones de las *Defensas*. Tambien maneja una parte de la logica de mover un *Enemigo*.

6.8. Turnos

La clase *Turnos* se ocupa de generar los *Enemigos* nuevos cada vez que el *Jugador* pasa un turno

6.9. Lector

El *Lector* se desarrolla a partir del patron de diseño *Facade*, un patron de diseño estructural que le proporciona una interfaz simplificada al Juego, para poder tener distintos tipos de lectores segun la extension de los archivos. Actualmente nuestro modelo solo recibe archivos *JSON* especificos, pero podria ser extensible a otros tipos de archivos.

6.10. Logger

El programa implementa un *Logger* que imprime por consola cada paso del juego. Este *Logger* utiliza el patron *Singleton* para poder ser una clase unica y accesible en cualquier parte del programa. Puede ser activada o desactivada en una sola linea de codigo.

6.11. Factories

Se utilizo el patron de diseño *Factory Method* para la construccion de los *Enemigos* y *Parcelas* desde el *Lector*.

7. Excepciones

CreditosInsuficientesError Se lanza cuando el jugador intenta gastar mas creditos de los que dispone.

EnemigoInvalidoError Se lanza en *EnemigosFactory* al no leer un *Enemigo* valido.

ParcelaInvalidaError Se lanza en el *ParcelasFactory* al no leer una *Parcela* valida.

NoSePuedeLeerElMapaError Se lanza cuando el *Lector* no puede generar el *Mapa* correctamente.

NoSePuedeLeerEnemigosError Se lanza cuando el *Lector* no puede generar los *Turnos* correctamente.

ParcelaNoPuedeUbicarError Se lanza cuando se intenta ubicar algo invalido en una *Parcela*.

TurnoInvalidoError Se lanza cuando se intenta agregar un *Enemigo* a un turno invalido.

8. Diagramas de secuencia

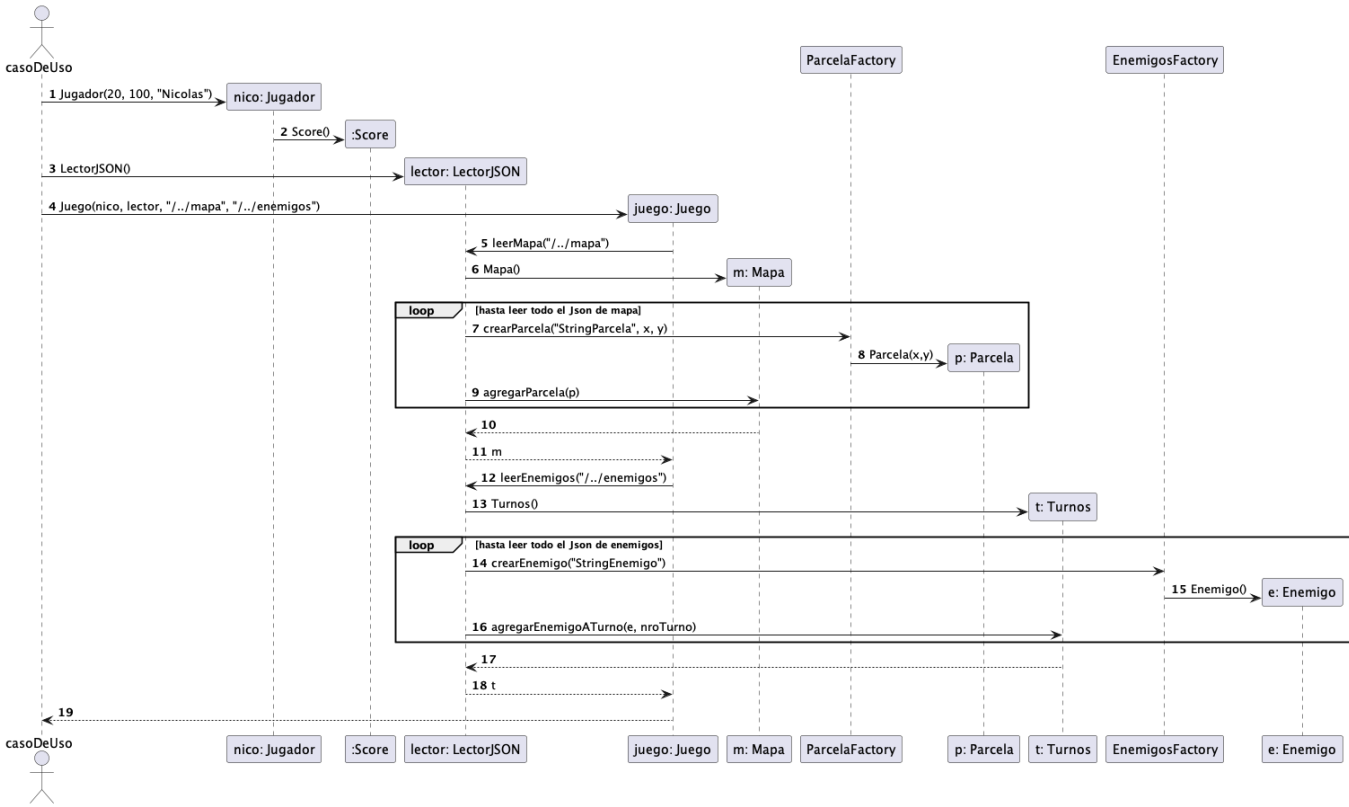

Figura 13: Construccion del Juego

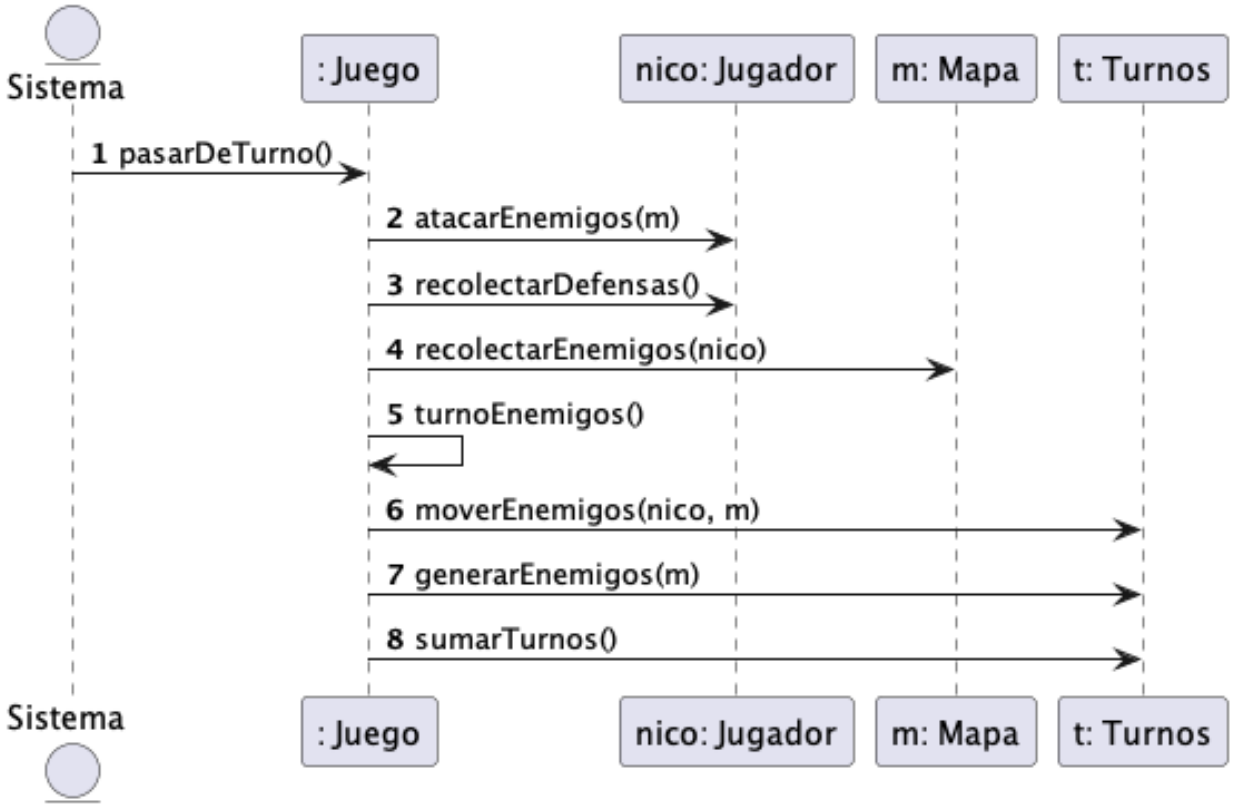

Figura 14: Pasar de turno

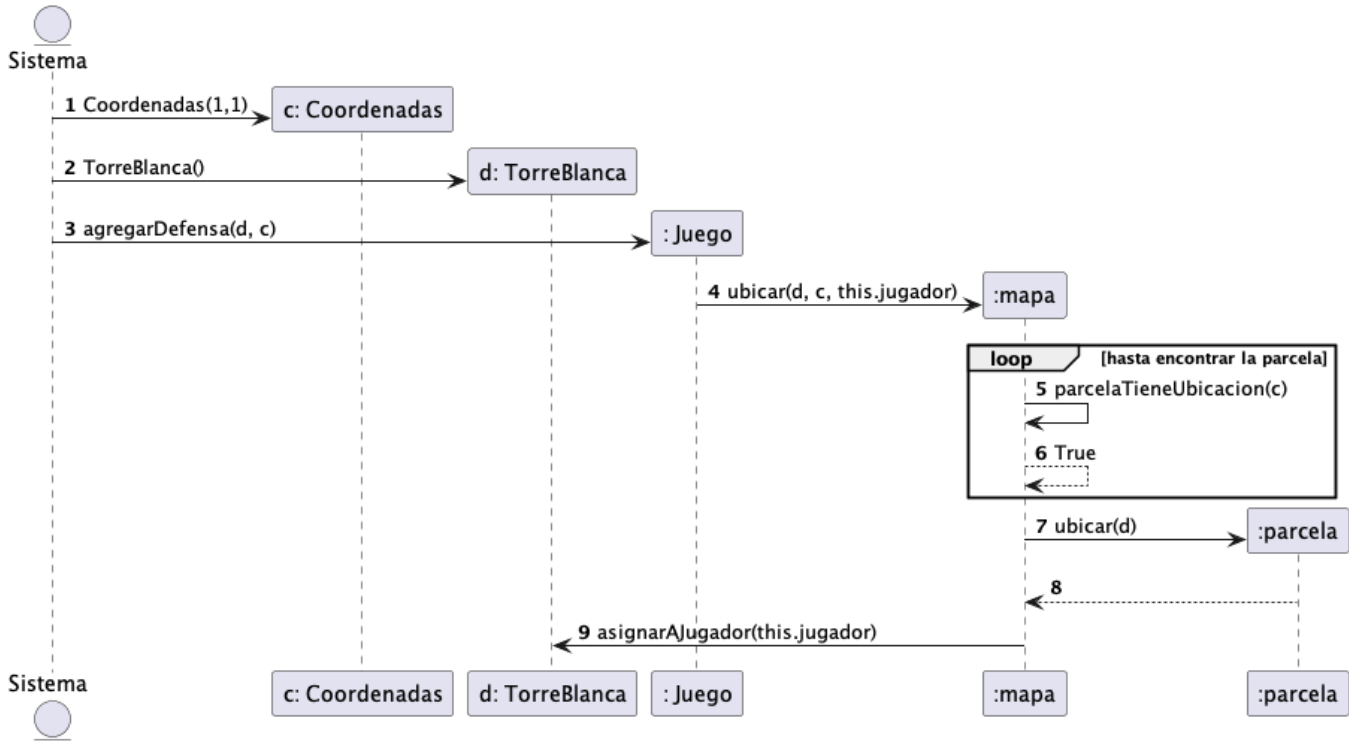

Figura 15: Agregar una Defensa al Juego

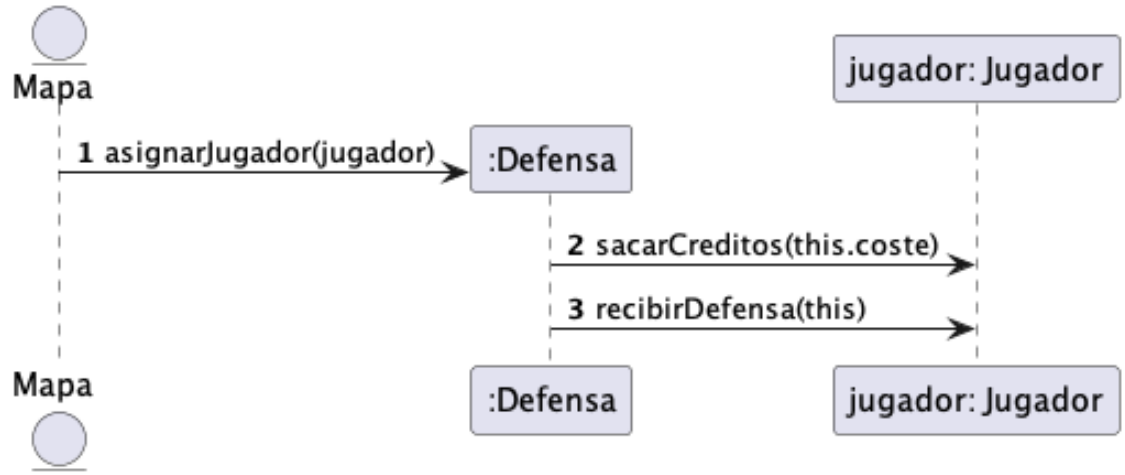

Figura 16: Asignacion de Defensa a un Jugador

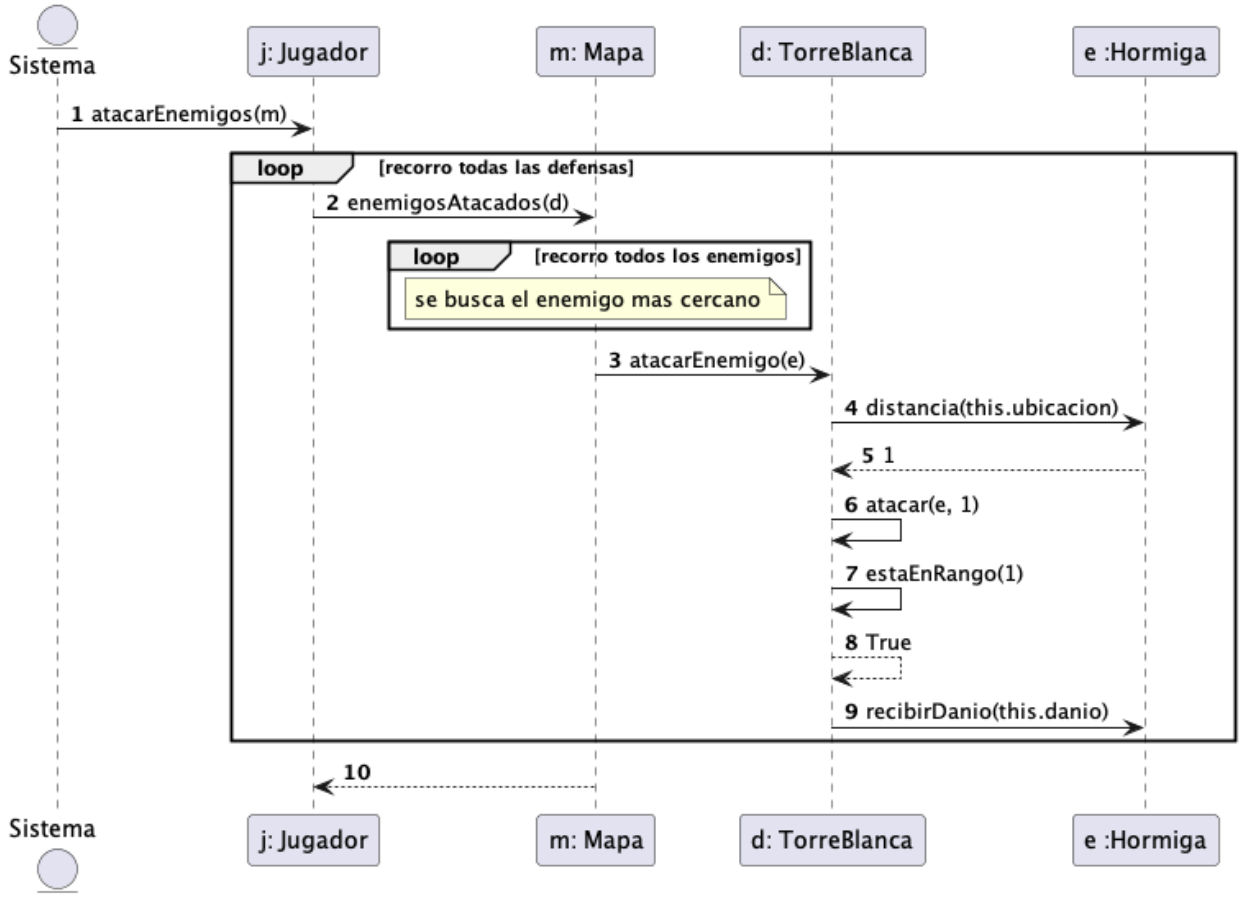

Figura 17: Defensas de un jugador atacan a los Enemigos

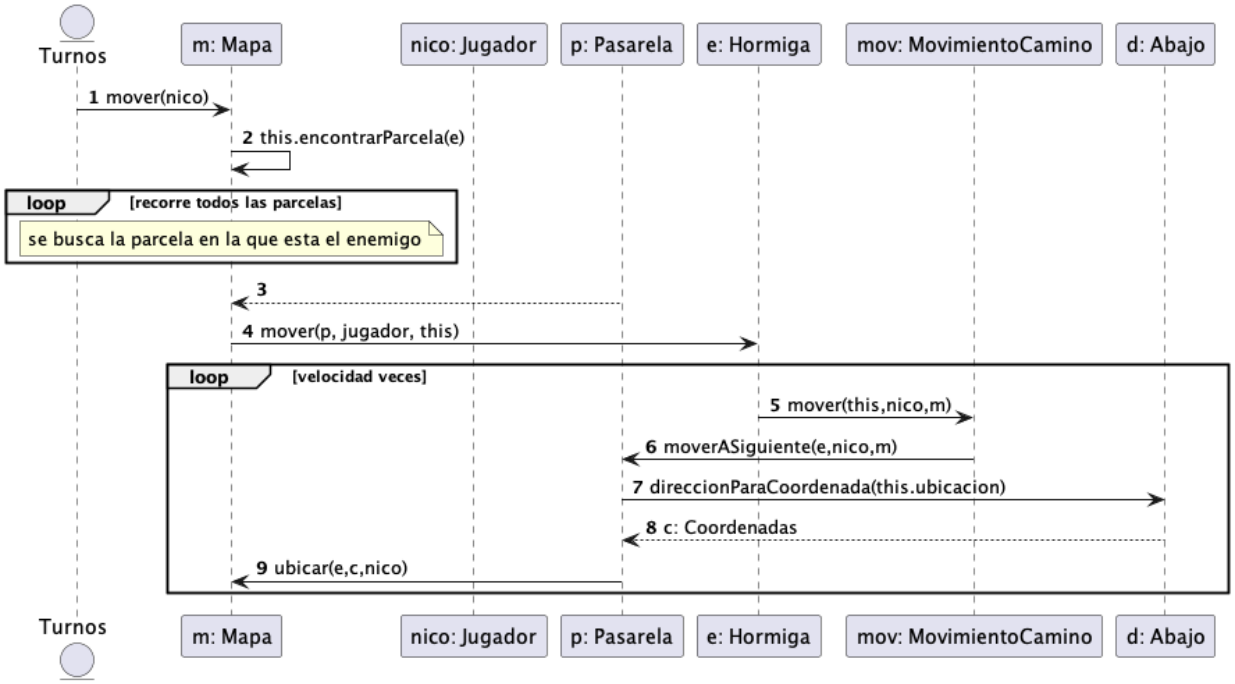

Figura 18: Movimiento de una Hormiga por el camino

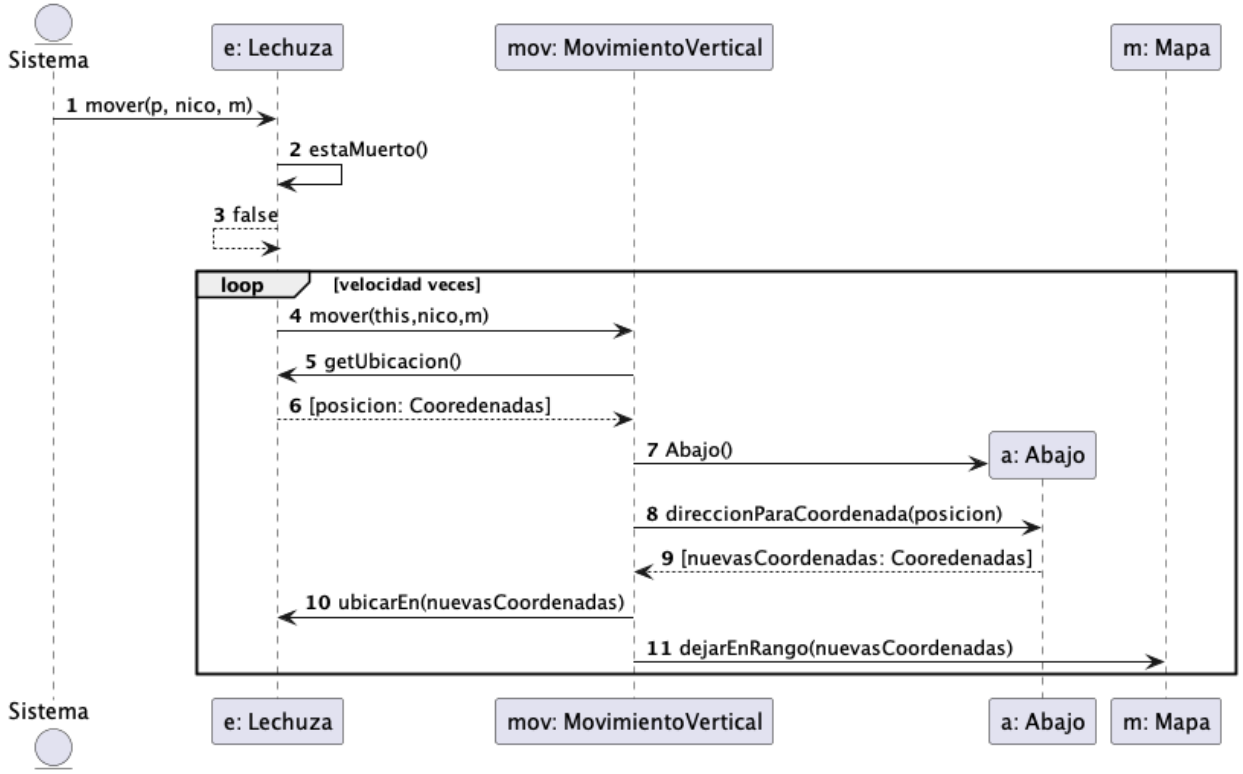

Figura 19: Movimiento de una Lechuza fuera del camino

9. Diagrama de estados

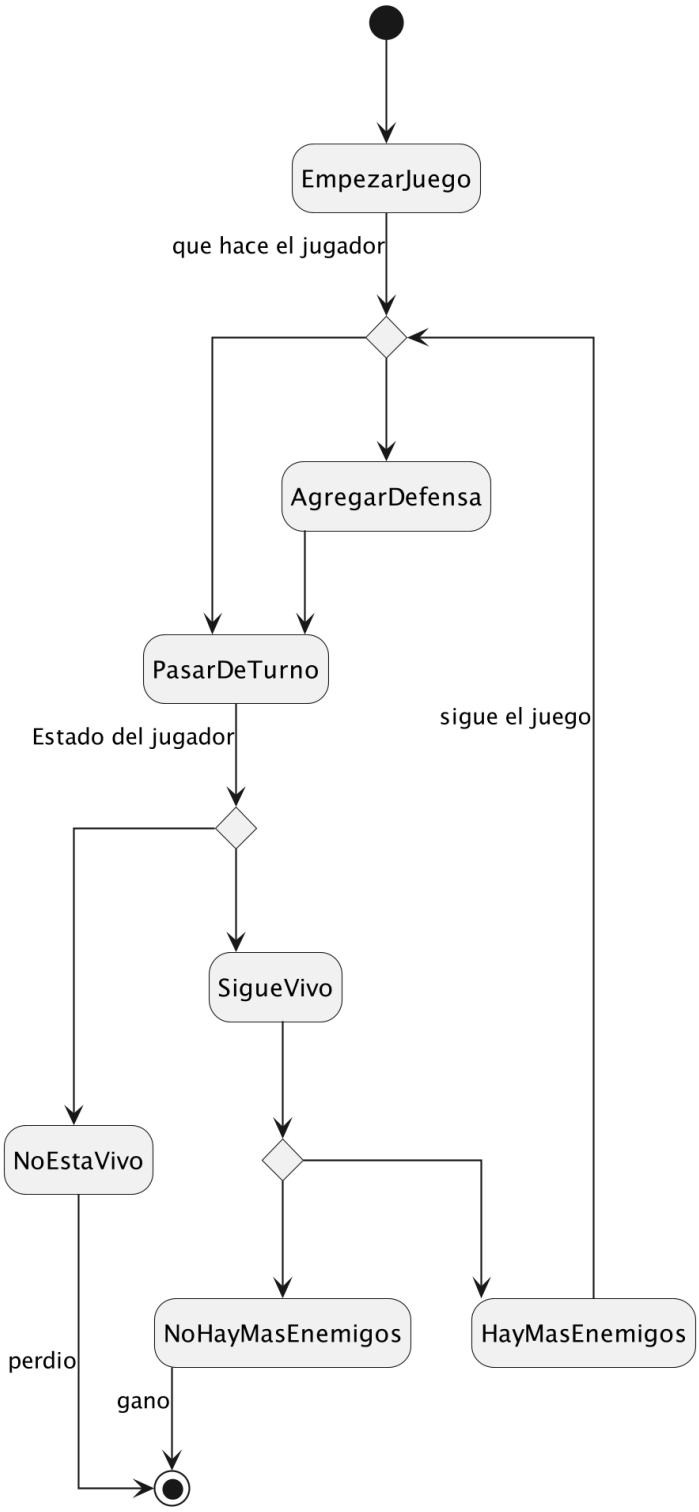

Figura 20: Estados del Juego