Trabajo Práctico N°2 — Java

[7507/9502] Algoritmos y Programación III
Curso 1
Primer cuatrimestre de 2023

Nombre y Apellido	Número de padrón	email
AMUNDARAIN, Tomas	108155	tamundarain@fi.uba.ar
SENDRA, Alejo	107716	asendra@fi.uba.ar
MARIANETTI, Daniel Agustin	106256	dmarianetti@fi.uba.ar
OSAN, Martín	109179	mosan@fi.uba.ar

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación de un sistema para un juego al que se lo llamó Tower Defense en java utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso.

2. Supuestos

Para el armado del modelo se tomó como supuesto que el mapa siempre va a ser cuadrado y que siempre hay un camino con más de una pasarela en el mapa.

3. Detalles de implementación

En esta sección se explicará con algunos ejemplos del modelo algunos conceptos que se vieron en la materia con sus respectivas justificaciones.

3.1. Defensas

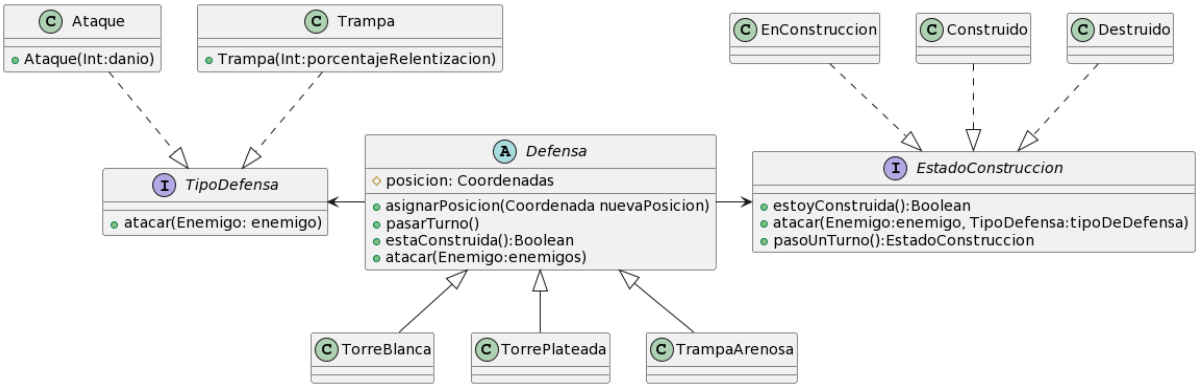

Figura 1: Detalles sobre la implementación de las defensas en el juego.

Para esta parte del modelo se utilizó una clase madre (Defensa) de la que heredan todas las defensas que existen hasta el momento. Además, se optó por separar en interfaces el tipo de defensa que es, ya sea el tipo que ataca y hace daño a los enemigos o del tipo que causa efectos a los enemigos, y su estado de construcción, ya que dependiendo del estado en el que se encuentre y su tipo, la defensa va a tener un comportamiento determinado. Observar que el comportamiento en el que va cambiar dependiendo de su estado y tipo es en el ataque de la defensa.
Para implementar el estado de construcción se utilizó el patrón State.

El uso de este patrón nos da la ventaja de que la defensa se vuelve mucho más open-close que si la lógica de su estado de construcción y su tipo estuviese en la defensa. Por lo que si en próximas actualizaciones se quiere que una defensa tenga un nuevo estado u otro tipo de ataque solo basta con crear nuevas clases que implementen EstadoConstruccion y TipoDeDefensa y agregarlas a la defensa.

3.2. Parcelas

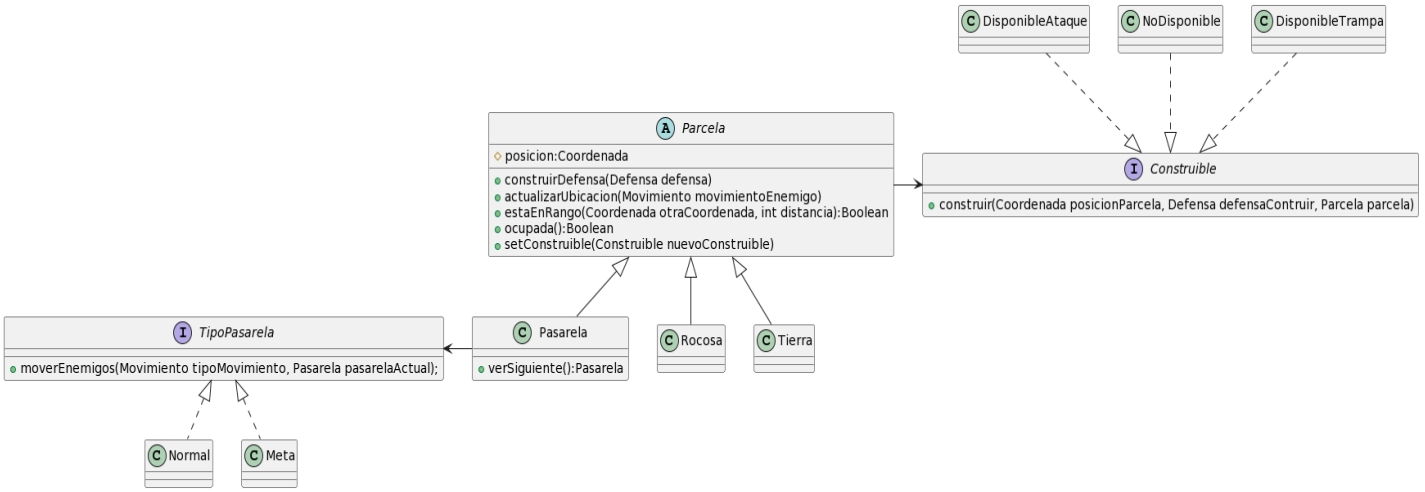

Figura 2: Detalle de implementación de las parcelas del juego.

Para implementar las parcelas del juego se definió una clase madre (Parcelas), que al igual que en el caso de las defensas, los diferentes tipos de parcelas heredan los atributos y métodos de Parcela. Por otro lado, se aplicó el patrón Strategy al implementar la interfaz Construible, debido a que dependiendo del tipo de parcela que sea es el tipo de defensa que se le puede construir y además de si está o no ocupada por otra defensa.

También en las pasarelas, se le delega el movimiento de los enemigos sobre la misma a TipoPasarela, ya que si la pasarela es la Meta el enemigo cuando se mueve debe aplicarle daño al jugador y luego debe desaparecer del juego.

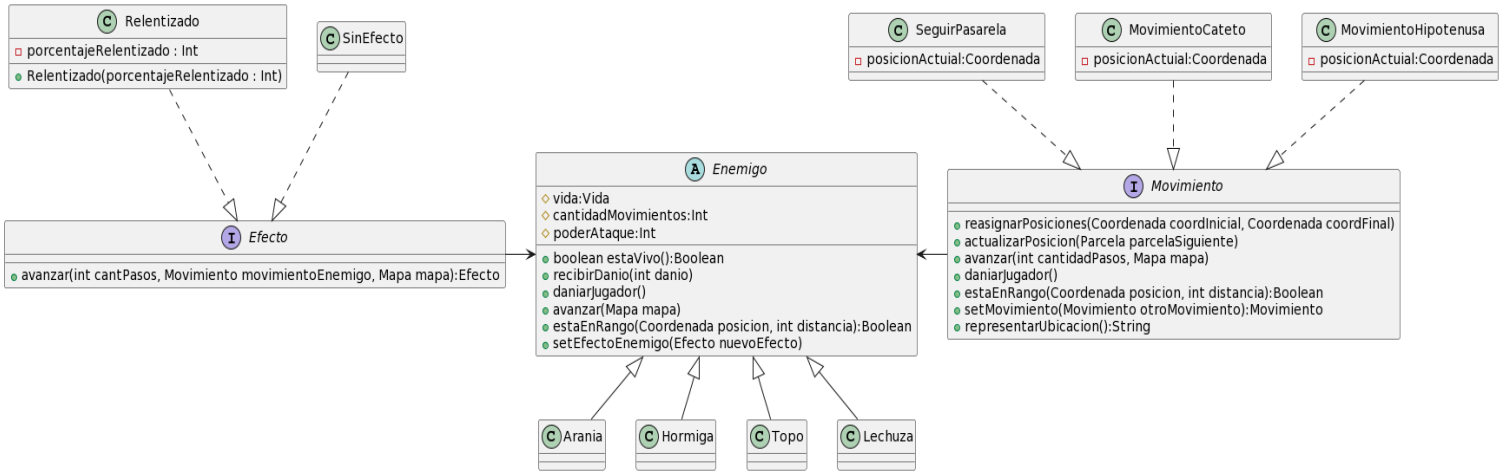

Figura 3: Detalles de implementación de los enemigos del juego.

3.3. Enemigo

En la implementación del enemigo también se usaron los patrones Strategy (en Efecto) y State (en Movimiento). Esto se debe a que el método avanzar() depende de que si el enemigo tiene en ese momento efectos o no y en la forma en que se mueve el enemigo. Un ejemplo es la Lechuza que primero tiene un movimiento MovimientoCateto y al perder una cierta cantidad de vida cambia su forma de moverse a MovimientoHipotenusa.

Esto nos da la ventaja de que si se desea el día de mañana cambiar la forma de moverse de un enemigo por otra nueva, solo hay que crear un nuevo tipo de movimiento y crearle toda la lógica. La misma situación ocurre con el Efecto del enemigo que también da la misma ventaja. Como consecuencia de esta ventaja se logra un bajo acoplamiento entre Enemigo y TipoDeMovimiento y Efecto.

3.4. MVC con Observer

Hacemos uso del patrón MVC para establecer la comunicación entre el modelo y la GUI, en resumen la idea de esto es poder separar responsabilidades y reducir el acoplamiento entre las distintas capas de la solución haciendo que las vistas detecten cambios en el modelo sin que el mismo tenga que explícitamente con la vista. Para lograr esto implementamos MVC con el patrón Observer, la idea es que las partes del modelo observadas reciban la orden de comunicar su cambio de estado en caso de haber sucedido a cada uno de los observadores (vistas) para que los cambios se vean reflejados en nuestra interfaz.

3.4.1. Creador de juego

Como se ve en el diagrama de secuencia 3.4.1, el botón inicio de juego funciona como controlador, que al hacer click en el mismo causa que se inicie la secuencia de crearJuego (mensaje que recibe de forma estática el CreadorJuego).

3.4.2. Vista vs Visualizadores

Para empezar, aclaramos que si bien por convención del patrón MVC sería más propio que las Vistas sean en sí cada una de las pantallas o parte de las pantallas que se muestran, por un tema de nomenclatura simplemente preferimos dejar los nombres de esta manera porque nos pareció más claro. En nuestro caso entonces, cada VistaX es en realidad un Observer que será notificado por los elementos del modelo pertinentes y comunicará estos cambios (más adelante explicaremos cómo) a los Visualizadores que se encargará de renderizar y organizar los datos en la GUI.

3.4.3. Visualizadores

Debido a la naturaleza de JavaFx de tratar la interfaz gráfica como un plano en que se superponen elementos visibles, decidimos separar los distintos componentes de la interfaz que funcionan de manera independiente y tener un visualizador padre que organice y renderize estos elementos ya configurados, en nuestro caso es VisualizadorMapa. Para poder lograrlo, como visto en la figura 5, al inicializar cada una de las vistas, se pasa por parámetro en el constructor al VisualizadorMapa a cada uno de los Visualizadores que a su vez se pasa como parámetro a cada una de las vistas. De esta manera, cuando una vista recibe una notificación de actualización de parte del modelo, comunicará al respectivo visualizador el cambio pasándole información relevante sobre la entidad cuyo estado cambió, el visualizador modificará como quiere mostrar la información y comunicará a Visualizador mapa el resultado para que lo haga visible.

3.4.4. Vistas

Cada uno de los observadores creados funcionan de manera muy similar, usando la librería java.util.Observer y java.Util.Observable. Cada vez que llega un update de información del modelo, castea el elemento recibido a aquello que desea observar, pide la información pertinente y comunica a la vista los cambios sucedidos. Ver figuras 6, 7 y 8.

3.4.5. Agregar observadores y notificar

Resultó un problema para nosotros buscar la forma de agregar a los observadores en los observados, ya que, por la naturaleza del juego de ir creando los objetos a medida que se introducen

en el juego, nos es imposible tener una referencia al modelo actual del programa en cada visita para pedir los datos pertinentes (en realidad sería posible pero sería obtener toda la información mediante setters anidados). Es por esto que, como se ve en la secuencia de inicialización, se pasa a los distintos elementos encargados o de instanciar o de incluir en el juego (en este caso sólo para enemigos y defensas) los observers que se deben agregar a cada uno. Este es una instancia única de observer por el motivo descrito en el punto 3.4.3, ya que de otra forma se debería tener una misma referencia a VisualizadorMapa en todos los observers.

Para notificar se creó un método notificar() en juego que se encarga de delegar a la entidad que contiene tanto las torres como enemigos, EstadoJuego, para que pida a los objetos que notifiquen a sus observadores.

3.5. Juego

La implementación de la clase Juego como patrón singleton derivó de la necesidad de modificar el estado tanto del jugador (recibir daño, recompensas, etc) como del mapa (destrucción de torres) en base al comportamiento de los Enemigos. A su vez permite modificar el estado del juego con mayor versatilidad, permitiendo modificar partes del mapa sin modificar el resto de las entidades que forman parte del juego o modificar la generación de enemigos gracias a la inyección de dichas entidades.

3.5. Visibilidad de los enemigos según el tipo de defensa

```
public interface TipoDeDefensa {
    2 implementations  tomy07417
    void atacar(ArrayList<Enemigo> enemigos, Coordenada coordenada, int rangoAtaque);

    1 usage  2 implementations  tomy07417
    boolean accept(Visitor enemigo);
}
```

```
public interface Visitor {
    1 usage  2 implementations  tomy07417
    boolean esVisiblePara(Trampa trampa);
    1 usage  2 implementations  tomy07417
    boolean esVisiblePara(Ataque ataque);
}
```

Para la visibilidad de los enemigos según el tipo de defensa se utilizó un double dispatch, ya que por ejemplo en el caso de la lechuza puede ser atacada por las torres pero no le afecta la trampa arenosa. Para implementar esta solución también se tuvo que utilizar el patrón Visitor. Este tipo de solución presenta una desventaja, la cual es que el modelo no queda tan open/close ya que al agregarse otro tipo de defensa también hay que agregar otro método en la interfaz Visitor para el nuevo tipo. Sin embargo, por la cantidad de tipos que hay y al ser una solución que se podía hacer de forma rápida se optó por esta opción para poder dedicarse a otras cuestiones, si en un futuro la cantidad de tipos se hace más grande se tendrá que cambiar esta solución por otra que sea más open/close.

3.6. Coordenadas

En esta clase se implementó un getter que devuelve las componentes de sus ejes. Sabemos que hay una pequeña violación de encapsulamiento y que eso no es correcto pero solo se utilizó este getter para hacer el cálculo de las parcelas por las que tiene que pasar un enemigo según su movimiento y como era una solución rápida se optó por la misma para poder dedicar más tiempo a cosas que tenían un nivel más alto de importancia y necesitaban más tiempo.

4. Diagramas de clase

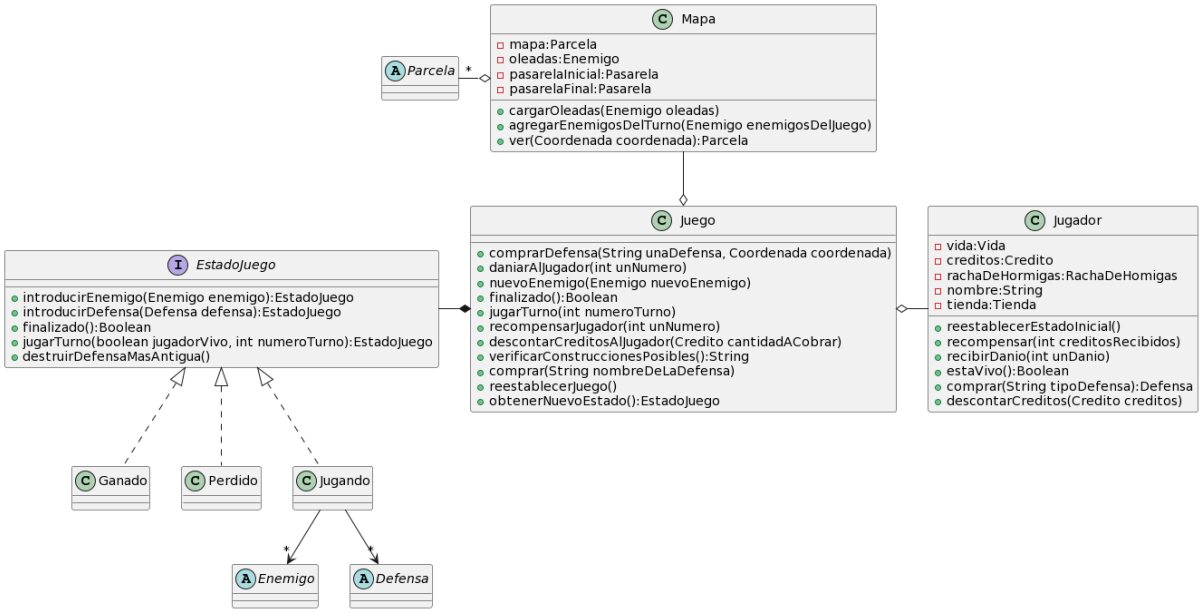

Figura 4: Esqueleto del modelo.

En este diagrama se muestran las relaciones principales del modelo, algunas clases como Enemigo que se mostró de forma específica cómo está formado no se mostraron sus atributos y métodos.

5. Excepciones

PasarelaInexistente: controla que el enemigo se mueva por parcelas que no sean nulas de hacerlo se arroja la excepción.
NoHayInicial: en caso de que se quiera establecer un camino en el cual ninguna pasarela se encuentra en el borde, por ende no hay un punto de entrada determinado
NoHayCamino: en caso que el json ingresado no tenga pasarela alguna

6. Diagramas de secuencia

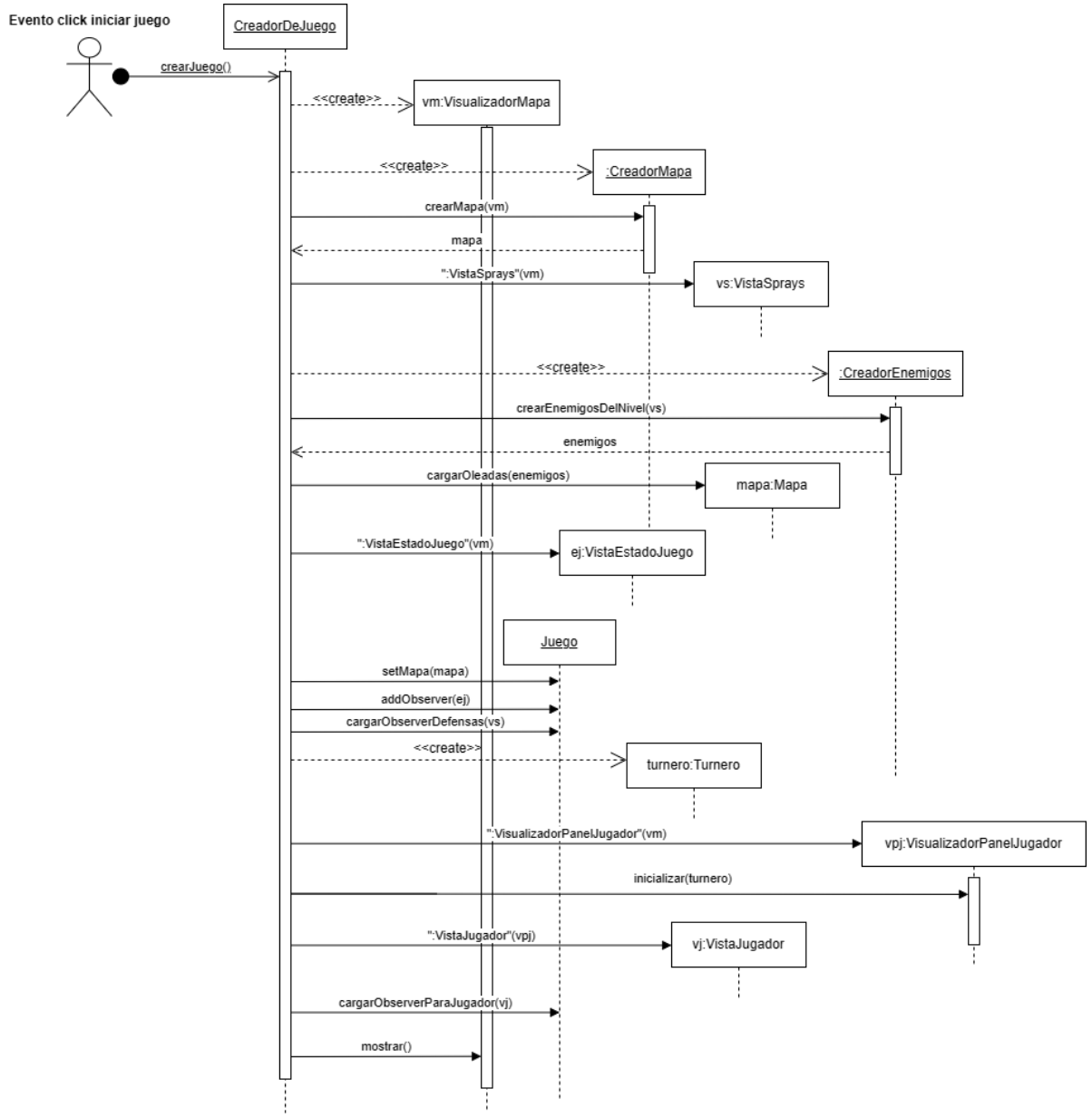

Figura 5: Creación de juego.

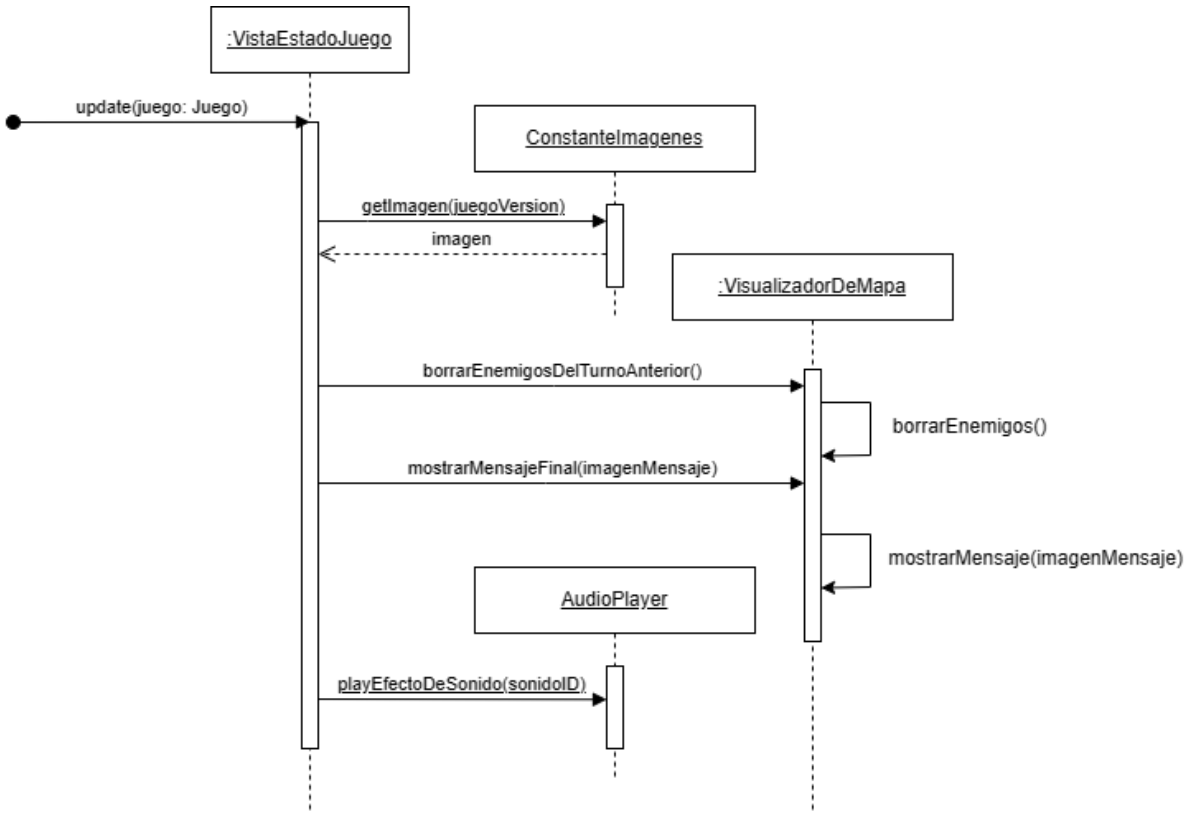

Figura 6: Vista estado juego

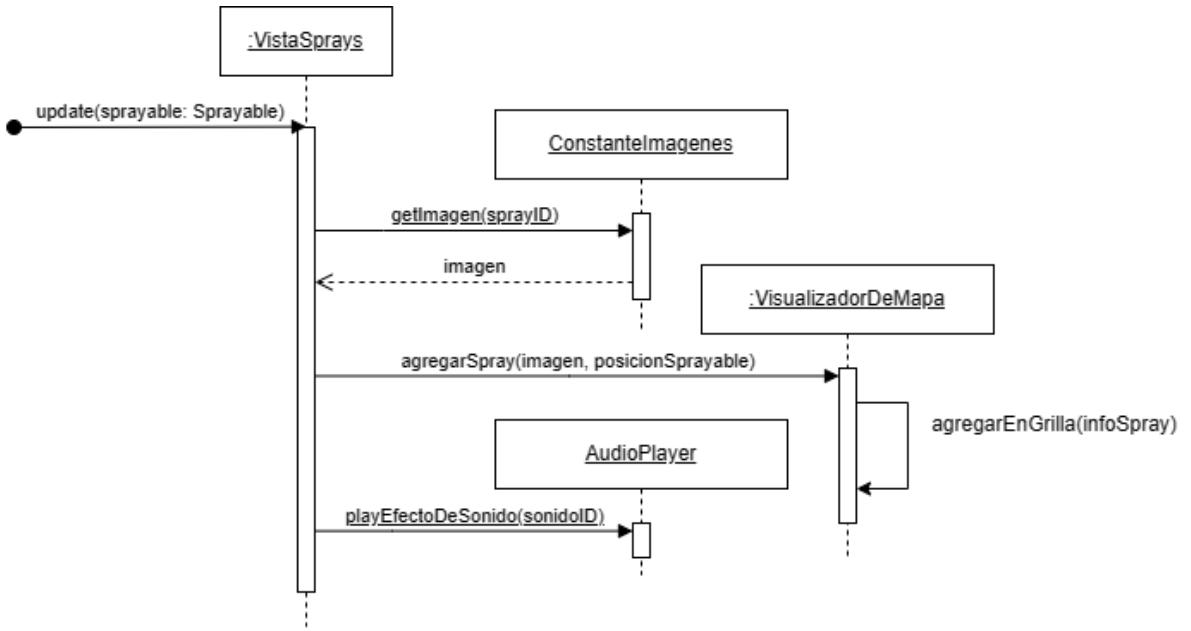

Figura 7: Vista sprays

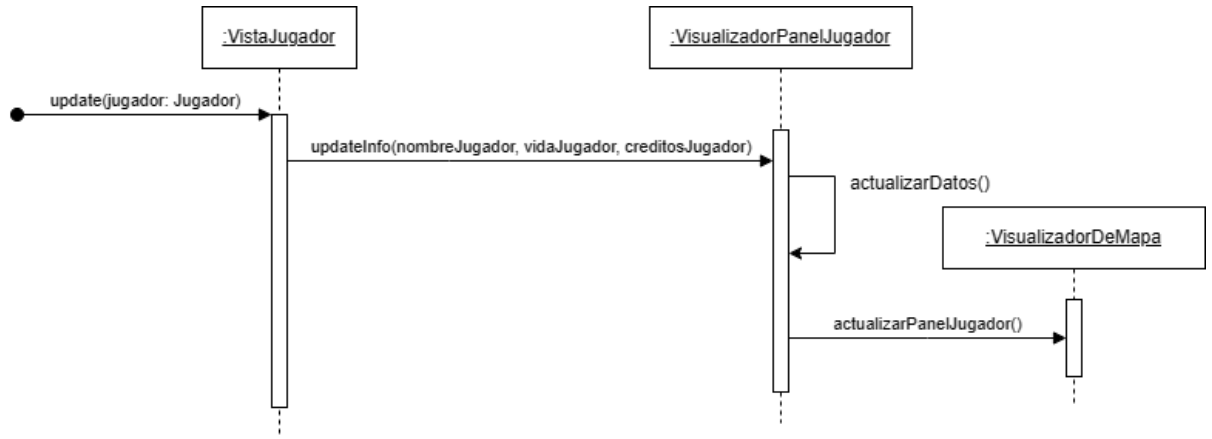

Figura 8: Vista jugador

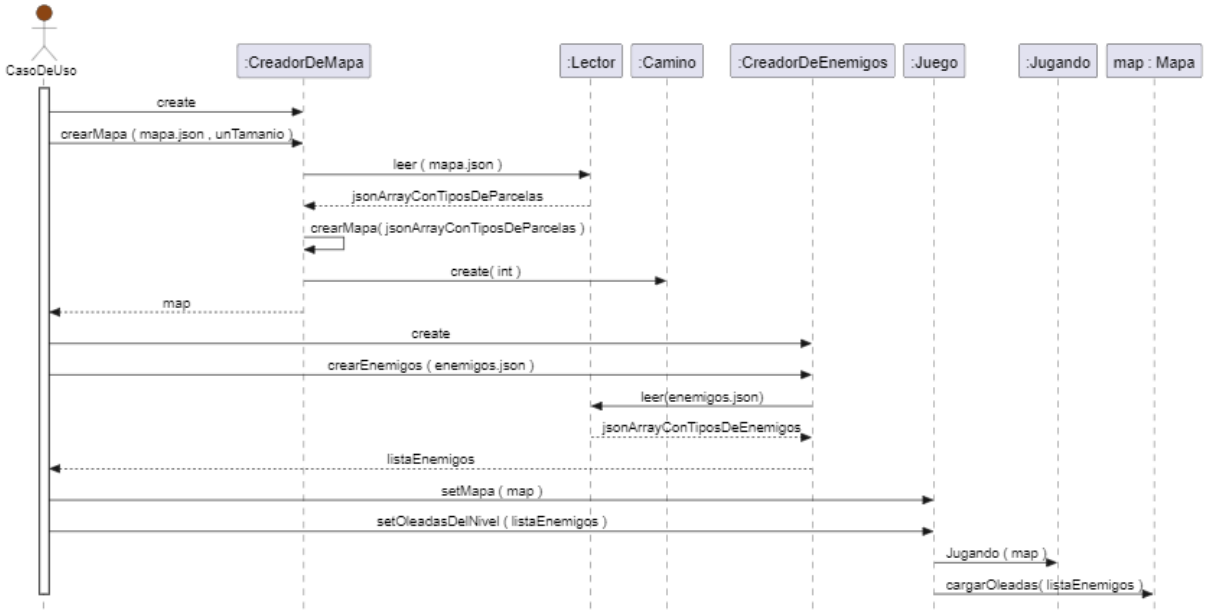

Figura 9: Ejemplo de creación de un nivel

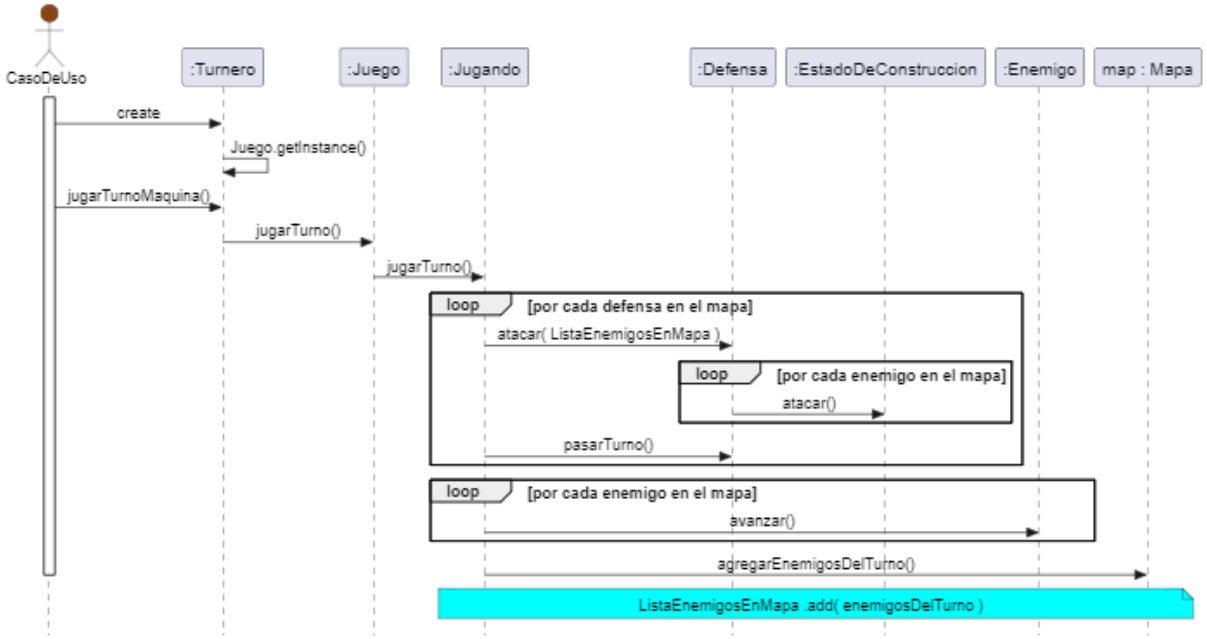

Figura 10: Ejemplo de el paso de los turnos y entidades con las cuales interactua

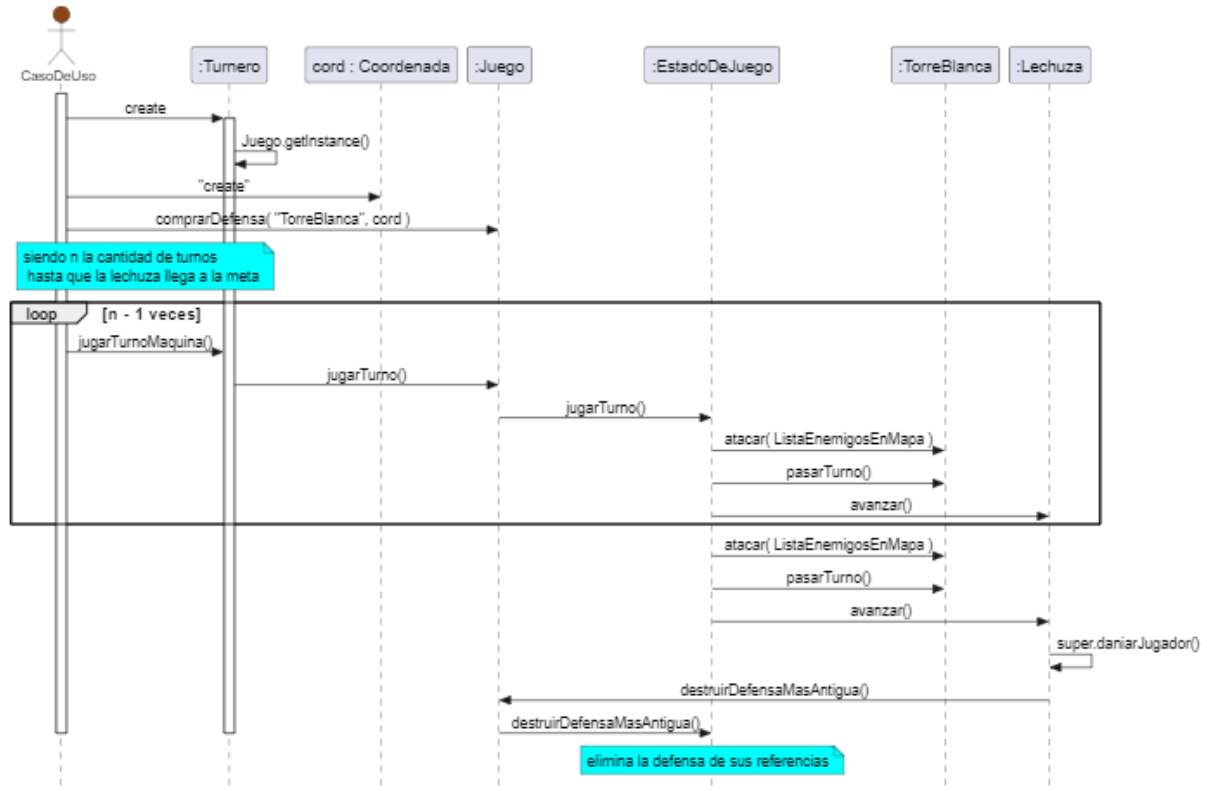

Figura 11: Destruccion de una torre al llegar, la lechuza, a la meta