FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Algo Defense

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

Tutor: Maia Naftali

Nota Final:

Alumno	Padrón	Email
Cabrera, Isaías Augusto	108885	iacabrera@fi.uba.ar
Llosas, Nicolas Pablo	105397	nllosas@fi.uba.ar
Marcelo, Origoni	109903	morigoni@fi.uba.ar
Exner, Tomas	109946	texner@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 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. La aplicación debe estar completa, incluyendo el modelo de clases, sonidos e interfaz gráfica. La aplicación deberá ser acompañada por pruebas unitarias e integrales y documentación de diseño

2. Supuestos

1. El rango de la torre es en forma de cuadrado y rango 3 representa un cuadrado

de 6 de lado. Con la torre en el centro

2. Torre ataca a solo un enemigo que será el primero que aparezca al iterar las posiciones en rango desde la esquina inferior revisando la x primero.

3. Camino esta en el orden correcto la primera pasarela que aparece es el inicio la ultima que aparece la ultima, todas en el medio siendo la secuencia del camino. Y el camino no puede tener menos de dos pasarelas

4. En el json de enemigos los turnos que aparecen no tienen turnos faltantes, no pasa del turno 3 al 5. Y también están en orden.

5. Las trampas generan un efecto permanente en los enemigos, a excepción del topo que al cambiar su velocidad deshace el efecto de la trampa

6. Las trampas no generan un efecto en enemigos de velocidad 1, y el efecto trunca la velocidad, redondea hacia abajo.

7. La lechuza no da créditos al jugador

8. El topo al no ser atacable por torres no puede morir.

9. Los enemigos solo atacan al jugador una vez, y mueren.

3. Modelo de dominio

El modelo de dominio para el juego de Algo Defense con interfaz gráfica en Java FX se compone de varias clases principales. Por la parte del modelo la clase "Juego" representa el juego en sí y es el encargado de mantener el estado actual de juego, guardar las cosas pertenecientes a este, el contexto, y además de manejar los cambios de estado en este, si bien el cuando se cambie de estado no es la responsabilidad de Juego. La otra pieza central es la clase " Mapa" esta clase es la encargada de mantener la información espacial de las Celdas/Parcelas, además de tener la responsabilidad de saber como accionar las defensas o enemigos como un conjunto, teniendo un observers para diferentes eventos. Por otro lado las estructuras defensivas que el jugador puede colocar se representan mediante las clases "Trampa", "TorreBlanca" y "TorreGris", cada una con sus propiedades específicas y definen mediante el accionar la acción a llevar a cabo, son las encargadas de encapsular la lógica de acción de una defensa además del costo de la construcción. Los enemigos similarmente están representados por las clases "Topo", "Lechuza", "Hormiga" y "Araña", quienes encapsulan la lógica de accionar de un enemigo, además de tener la responsabilidad de saber como acreditarse y como instanciarse, con el patrón prototype. Por último la última pieza central es el Jugador quien se encarga de mantener el estado del Jugador, delegando el tema créditos a un SistemaCreditos. La interacción de los controladores/ui con el modelo esta centralizada en AlgoDefense que funciona como un facade de Juego permitiendo ciertas acciones para evitar el manejo directo de los controladores de distintos componentes, y propagar menos los cambios que puedan ocurrir en el modelo.

4. Diagramas de clase

En los siguientes diagramas se presentan de forma general las diversas relaciones que presentan las clases que conformas nuestro modelo.

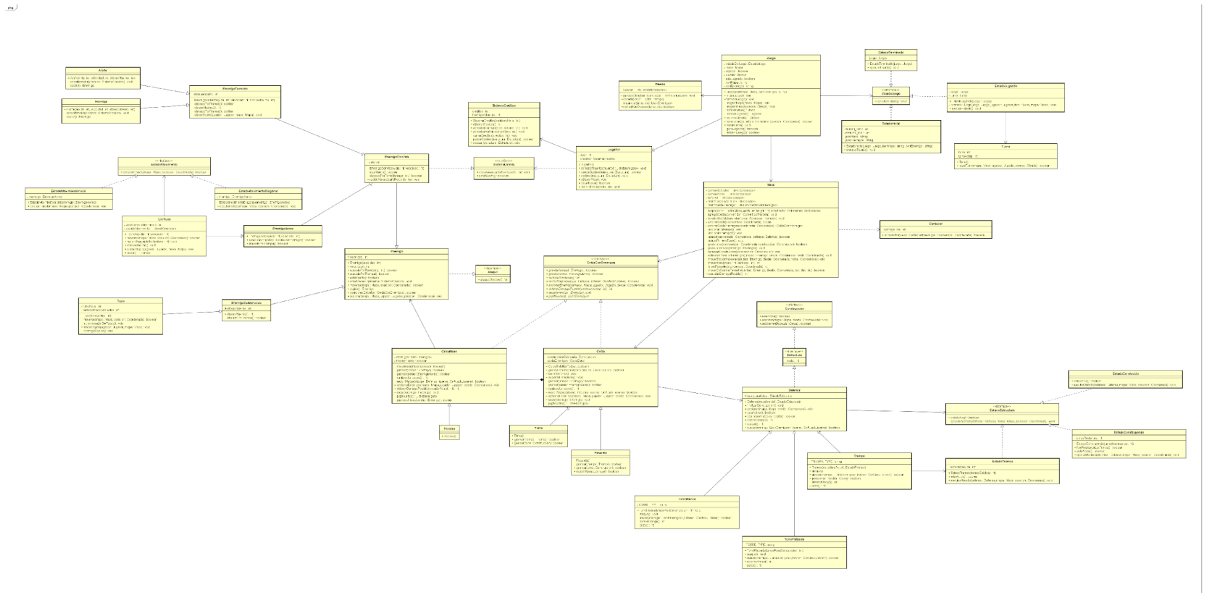


Figura 1: Diagrama general de clases

Es difícil hablar del modelo en general y por esto se va a hablar por partes, empezando con el flujo de estados en el juego se tendría el siguiente diagrama de clases

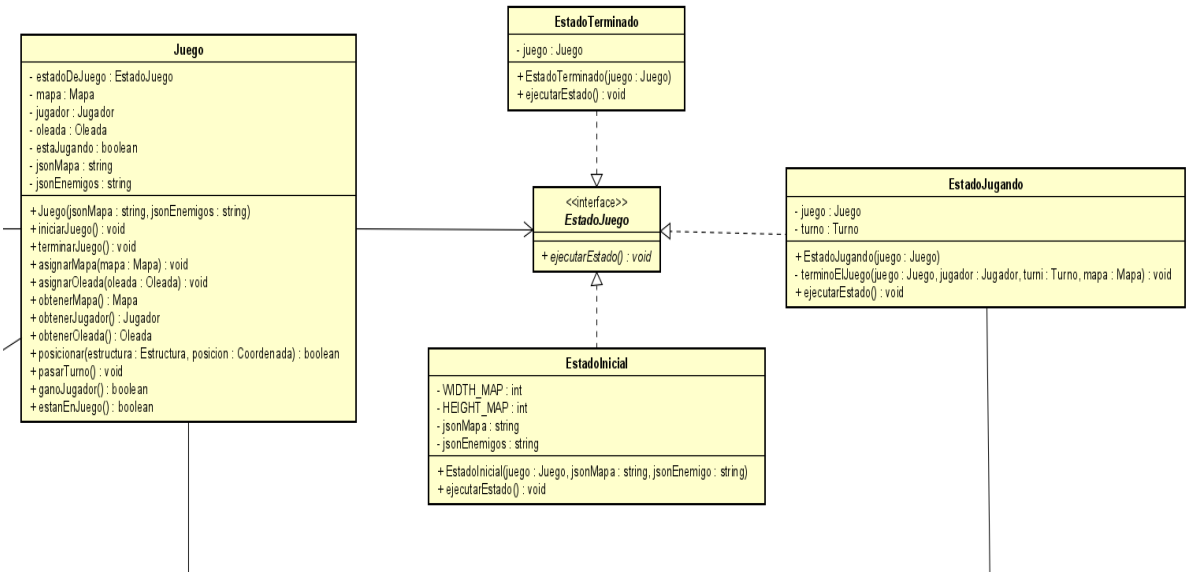


Figura 2: Diagrama de estados de Juego

Se observa la existencia de 3 estados posibles para juego, EstadoInicial que se encarga de inicializar el juego desde un json del Mapa y uno de enemigos. EstadoJugando que encapsula la lógica del turno, que se la delega al Turno, y se encarga del cuando es que se debería terminar el juego. Y EstadoTerminado que esta como un estado para poder mostrar resultados y poder volver al inicio de quererse. Aunque actualmente no tenga demasiada lógica al simplemente crearse un nuevo juego en este sentido.

Para adentrarnos mas en detalle veamos en especifico que es el contexto del juego:

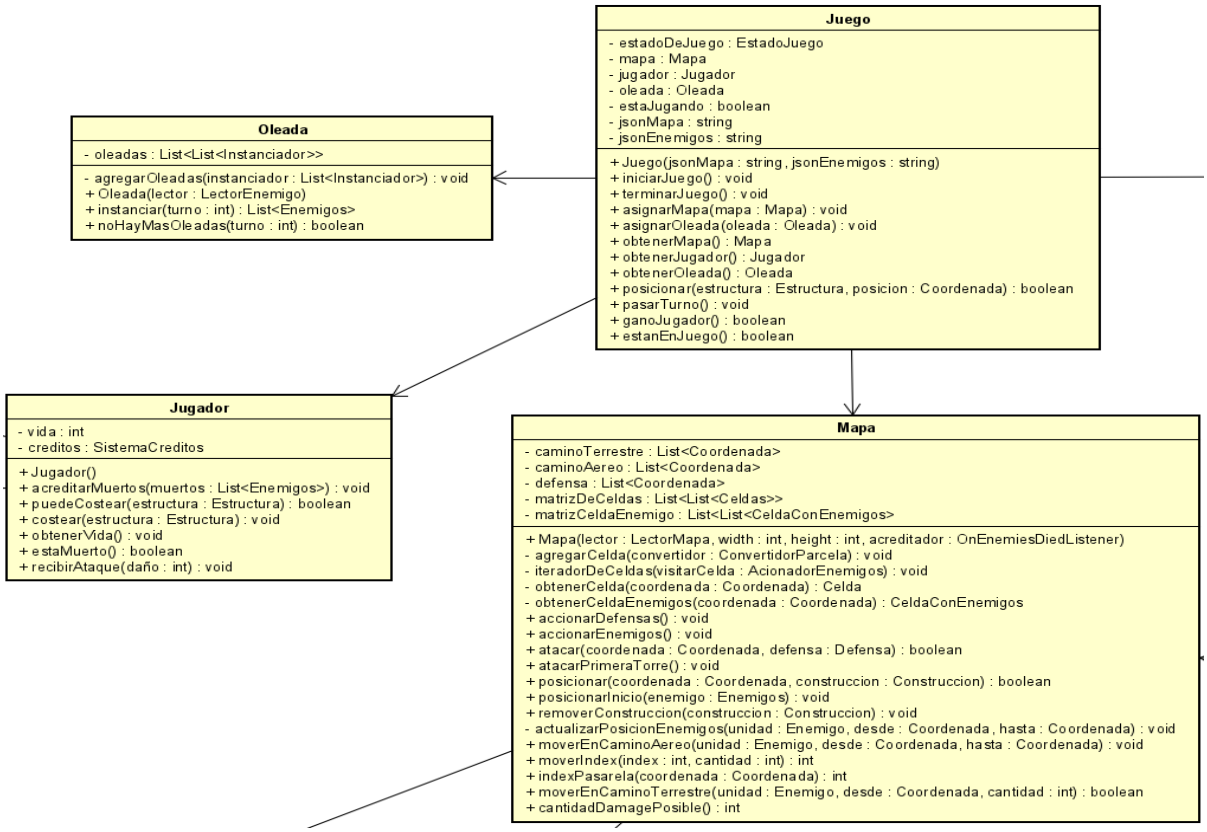


Figura 3: Diagrama de asociaciones de Juego

Se observa que el juego tiene 3 objetos como su contexto, a un Jugador, a un Mapa y una Oleada, que representaría a las oleadas. El pasarTurno seria delegado al Estado ejecutarEstado. Prosiguiendo se tendría a Mapa:

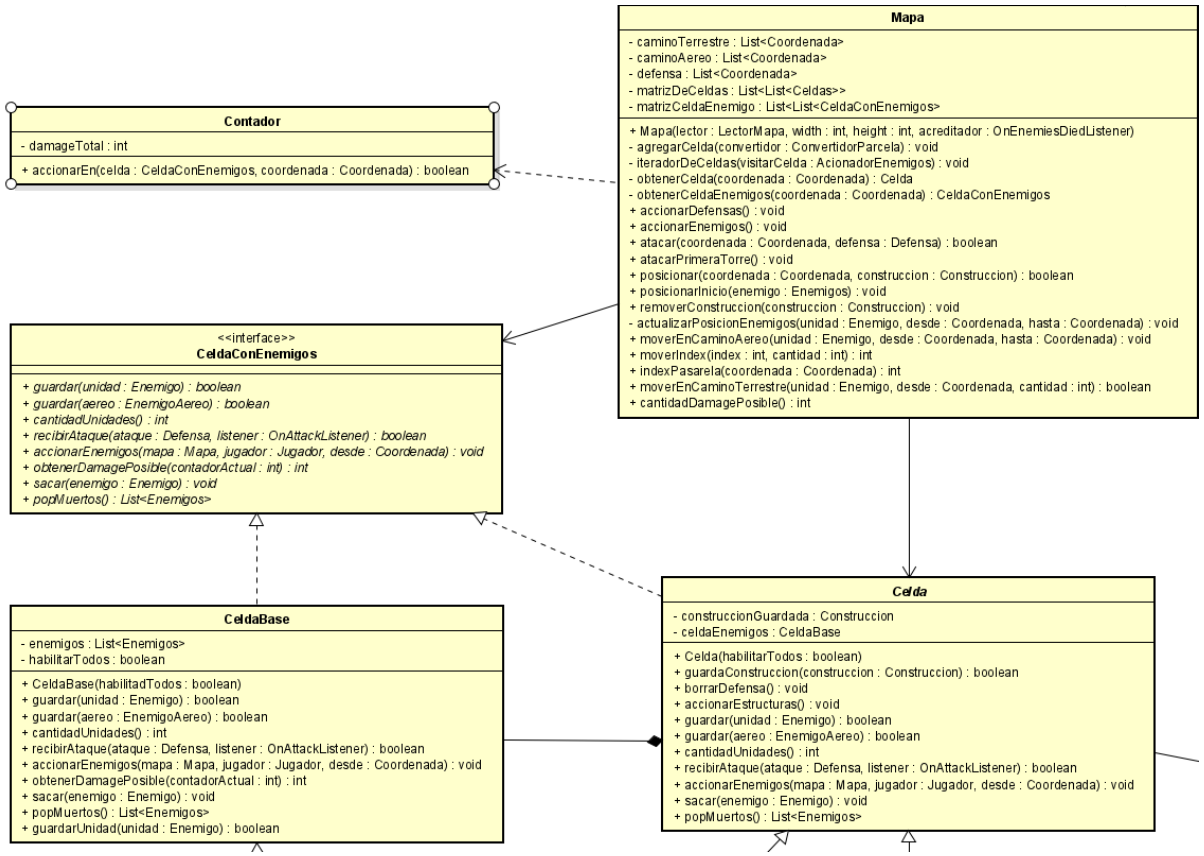


Figura 4: Diagrama de relación de Mapa con Celdas

Mapa separa por contrato entre CeldasConEnemigos y Celdas. La diferencia estando en que una puede albergar construcciones y la otra no. Ya que actualmente todas las celdas permiten el guardado de Enemigos en ellas, aunque sea solo Lechuzas, se observa el uso de una CeldaBase que define el como se maneja a los enemigos dentro de una celda, de la cual después Celda que define como se manejan las construcciones va a usar por medio de composición, cuya flecha no esta en el diagrama. Esta forma de hacer las cosas es proveniente de que no se puede usar herencia para no romper el principio de liskov, al agregar la funcionalidad extra de manejar construcciones. Y tampoco se puede poner todo en una clase al no solo romper el principio de single responsability, sino que también rompería el de segregación de interfaz, al no todas las celdas poder guardar construcciones. En mapa el accionar enemigos y accionar defensas delega a las celdas, quienes delegan a los enemigos/defensas correspondientemente el accionar. Esto debido a que hay variabilidad en el accionar de estos y es preferible esa lógica quede en estos.

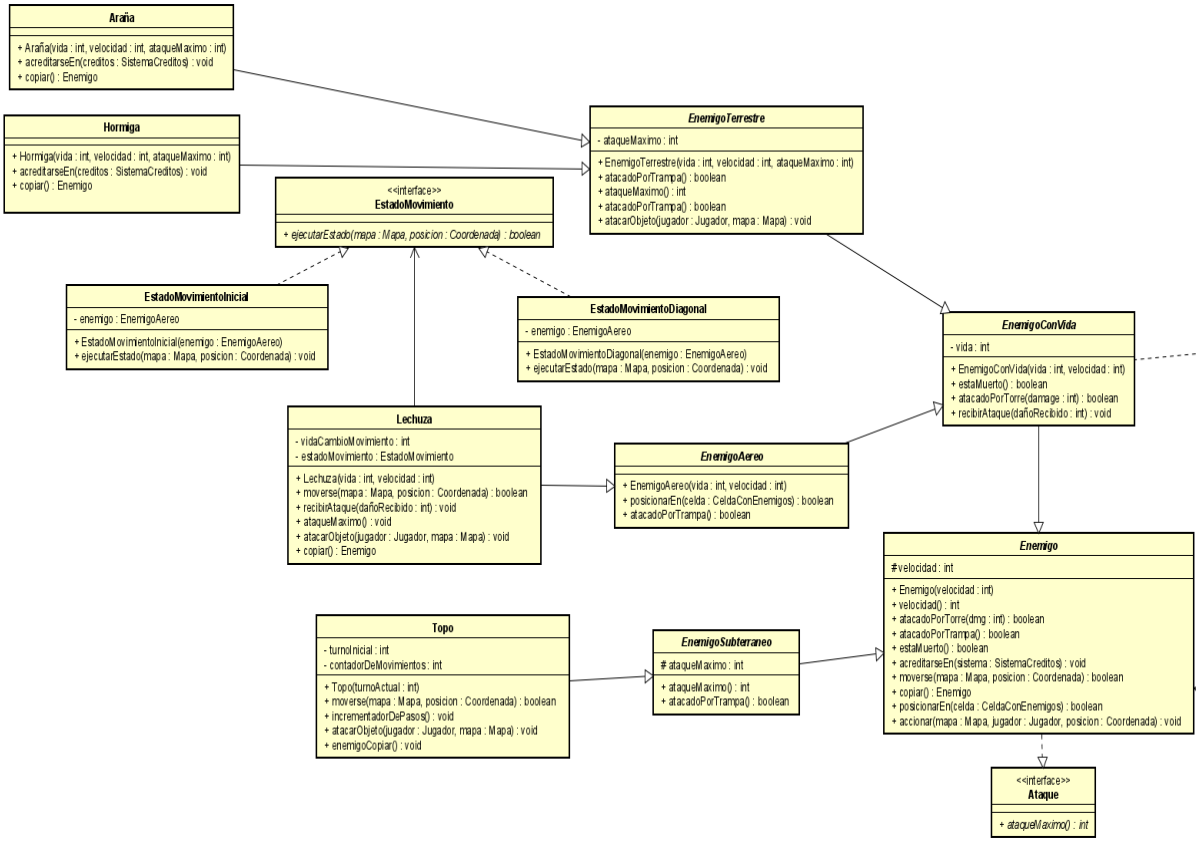


Figura 6: Diagrama de relaciones entre enemigos

Al accionar se le pasa la coordenada o posición para que esencialmente el Enemigo no tenga que saber de posiciones, similarmente:

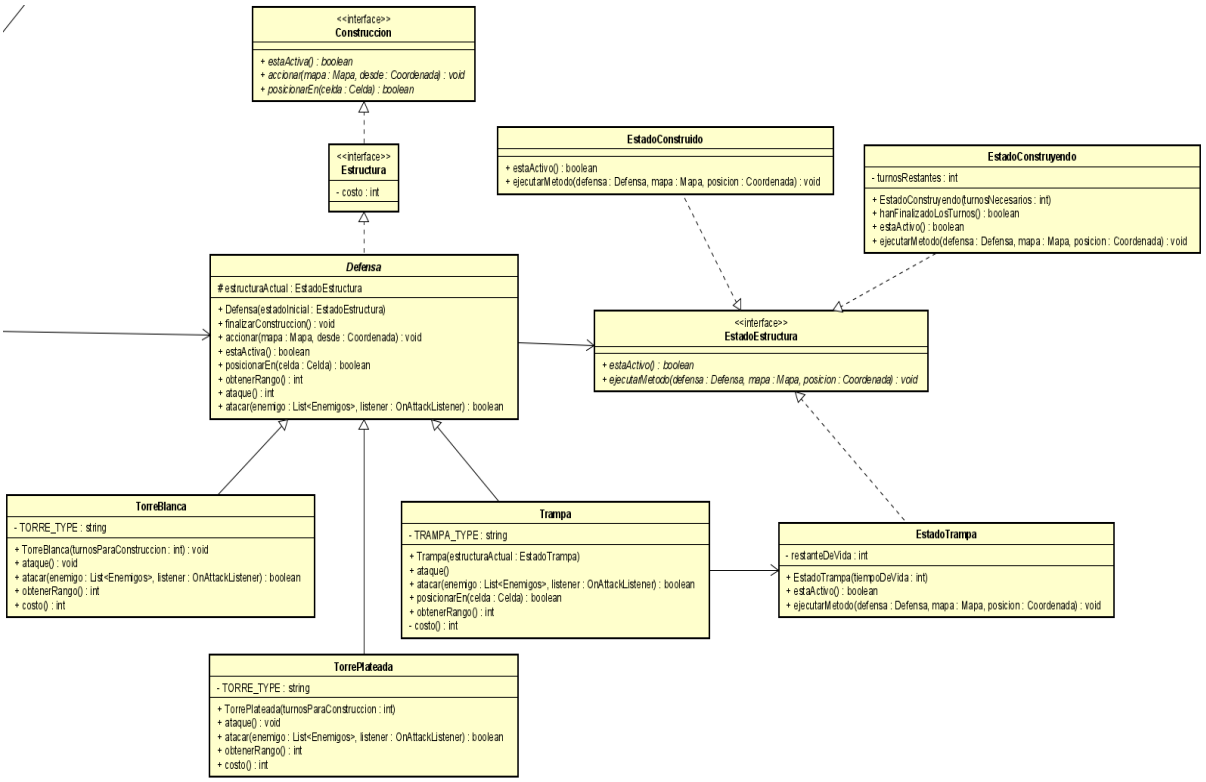


Figura 7: Diagrama de relaciones entre estructuras

La Defensa no guarda información sobre su posición, y simplemente acciona desde la posición que le pasen, en caso de que las defensas tengan que atacar en accionar. Se efectuara mediante el Mapa atacar. Tras la efectuación del ataque mapa notifica al acreditador de enemigos muertos, el Jugador, y si cambio la celda al observer de habitantes.

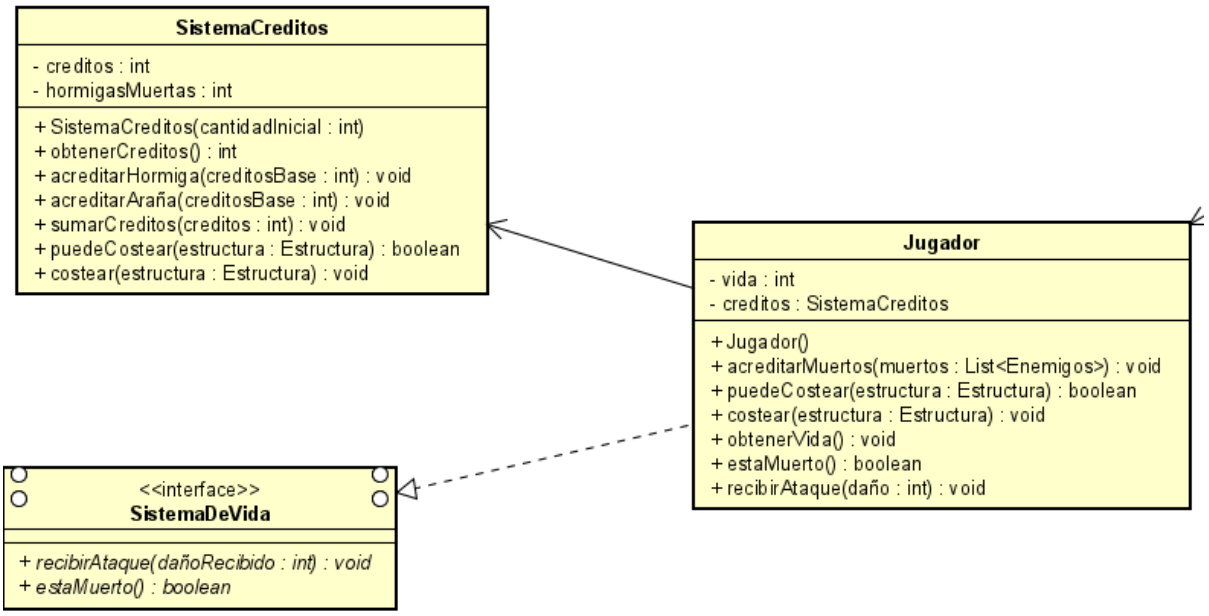


Figura 8: Diagrama de sistemas de Jugador

5. Detalles de implementación

5.1. Juego

En Juego, EstadoInicial es el encargado de inicializar o asignar al Juego el Mapa y Oleada según los json. El Jugador es creado por el Juego y asignado a si mismo en su creación. Esto para poder re iniciar el estado del Juego si se quisiese sin reiniciar al Jugador. Por otro lado EstadoJugando que es el estado que encapsula la lógica del juego, le delega la lógica del turno a Turno. EstadoJugando encargándose de definir y notificar a Juego sobre que el juego termino. Además Juego se encarga de delegar a Jugador el costeo de una defensa y a mapa su posicionamiento, a Jugador antes de decirle que costee le pregunta si puede. Esto para evitar que en el caso de que se efectué una acción que se deba revertir después, en el caso de que si se pueda costear pero no posicionar, o viceversa si se posicionar primero, por mas que se tire una excepción el Jugador quedaría en estado invalido tras la acción a menos se deshaga el cambio. Se podría asumir que siempre se va a poder costear y solo intentar posicionarla primero antes de costear. Que es lo que ocurre actualmente, pero para no hacer ese supuesto y asegurar no se hacen acciones invalidas es que se pregunta primero al Jugador en vez de ser mas asertivos. Por ultimo Juego guarda a lo que seria el observer del turno, pero no se encarga de notificar, esto lo haría el estado correspondiente, EstadoJugando.

5.2. Estado Juego

EstadoJuego es una interfaz creada con el objetivo de aplicar uno de los patrones de diseño conocido como “State”. En el mismo lo que hacemos es que según los diferentes estados de juego que se implementen esta interfaz, cada uno va a definir un método “ejecutar” con una funcionalidad diferente. Estos definiendo de ser necesario cuando cambiar de estado, y delegando a Juego este cambio, el cual lo validara también.

Como fue mencionado se cuentan con tres tipos diferentes de estados posibles, siendo estos Estado Inicial, Estado Terminado y Estado Jugando.

5.3. Jugador

La clase "Jugador" es responsable de gestionar la vida y los créditos del jugador en el juego. Implementa la interfaz "SistemaVida" y "OnEnemiesDiedListener" para recibir notificaciones cuando los enemigos son derrotados. Tiene propiedades para almacenar la vida del jugador y el sistema de créditos representado por la clase "SistemaCreditos".
La clase "Jugador" tiene interfaces en su interior tales como "modificacion_vida" y "modificacion_creditos" para permitir la observación de cambios en la vida y los créditos respectivamente.

El jugador comienza con una cantidad inicial de vida (20) y créditos (100). El Jugador le delega a los Enemigos el acreditarse en el SistemaCreditos, un double dispatch. También le delega el si se puede costear y el costeo a SistemaCreditos. El jugador puede recibir ataques que reducen su vida, y se verifica si está muerto en función de su vida restante.

La clase también tiene métodos para establecer observadores de cambios en los créditos y la vida del jugador. Estos observadores se notificarán cuando haya cambios en los valores correspondientes.

5.4. Turno

La clase "Turno" se encarga de gestionar los turnos de Algo Defense. Tiene una propiedad "turno" que almacena el número de turno actual, el cual se inicializa con el numero uno a la hora de crearlo.
La clase tiene un método "turnoActual" que devuelve el número del turno ya que solo existe una instancia de esta clase en todo el juego, por lo que a medida que este avanza, el número de turnos se incrementa en uno. Además, tiene un método principal "jugarTurno" que toma como parámetros al contexto del juego. En el turno se llevan a cabo diferentes acciones.
En primer lugar, se ejecuta el método "accionarEnemigos" del mapa, que

permite a los enemigos moverse y atacar al jugador. Si el jugador resulta muerto después de este ataque, se registra y se devuelve un valor booleano "false" para indicar el fin del juego.
A continuación, se ejecuta el método "accionarDefensas" del mapa, que permite a las defensas atacar a los enemigos en el mapa.
Después, se instancia una nueva oleada de enemigos utilizando el método "instanciar" de la clase "Oleada", pasando como argumento el número de turno actual. Estos enemigos se posicionan en el inicio del mapa utilizando el método "posicionarInicio" del mapa.
Finalmente, se notifica al mapa el inicio de un cambio en el turno utilizando el método "notificarInicioCambio".
El número de turno se incrementa en 1 al finalizar el turno. El método devuelve "true" para indicar que el juego debe continuar.

5.5. Oleada

La clase "Oleada" se encarga de gestionar las oleadas de enemigos de Algo Defense. La clase tiene un atributo "oleadas" que almacena una lista de listas de "Instanciador". Cada lista de "Instanciador" representa una oleada de enemigos.
La clase tiene un método privado "agregarOleadas" que se utiliza para agregar nuevas oleadas a la lista "oleadas". El método recibe una lista de "Instanciador" y la añade a la lista de oleadas. Cada Instanciador, se encarga de instanciar una cierta cantidad de veces a un enemigo, para esto se usa un patrón prototype, para simplificar el proceso, aunque también se podría ver de aplicar Factory pattern, se decidió no hacerlo ya que la opción discutida tendría un mayor incremento en la cantidad de clases por cada enemigo, cosa que se considero indeseable.
El constructor de la clase "Oleada" recibe un objeto "LectorEnemigo" que se utiliza para cargar las oleadas desde un archivo o fuente de datos externa. El constructor inicializa la lista de oleadas y, mediante el uso del "LectorEnemigo", agrega cada oleada a la lista utilizando el método "agregarOleadas".
Por otro lado, la clase posee un método "instanciar" que recibe el número de turno actual como parámetro. Este método se utiliza para obtener la lista de enemigos que se deben instanciar en el turno actual. Primero, verifica si no hay más oleadas disponibles para el turno actual utilizando el método "noHayMasOleadas". Si no hay más oleadas, devuelve una lista vacía de enemigos. Si hay oleadas disponibles para el turno actual, crea una lista vacía "enemigosNuevos" para almacenar los enemigos instanciados. Luego, itera sobre cada "Instanciador" de la oleada correspondiente al turno actual y utiliza el método "agregarInstanciasA" para agregar los enemigos instanciados a la lista "enemigosNuevos".

Al finalizar, devuelve la lista "enemigosNuevos" que contiene los enemigos instanciados para el turno actual.
Finalmente, la clase también tiene un método "noHayMasOleadas" que verifica si no hay más oleadas disponibles a partir del turno especificado. Compara el número de turno con el tamaño de la lista "oleadas" y devuelve "true" si el turno es mayor que el tamaño de la lista.

5.6. Mapa

La clase "Mapa" es de las más importantes ya que por ella pasa el flujo de la información por la que interactúan los actores que participan durante el juego. A continuación, se detalla con más precisión su estructura y funcionalidades:

- Interfaces:
  - La interfaz "OnEnemiesDiedListener" define un método "acreditarMuertos" para notificar cuando los enemigos mueren.
  - La interfaz "OnHabitantesChangedListener" define un método "cambio" para notificar cambios en los habitantes de una celda.
- Clase "Contador":
  - Esta clase implementa la interfaz "AccionadorEnemigos " y se utiliza para contar el daño total de las celdas durante la iteración.
- Atributos:
  - "caminoTerrestre": Una lista de coordenadas que representa la posición de los enemigos terrestres que caminan sobre el mapa.
  - "caminoAereo": Una lista de coordenadas que representa la posición de los enemigos aéreos que caminan sobre el mapa.
  - "defensas": Una lista de coordenadas que almacena las posiciones de las defensas en el mapa.
  - "matrizDeCeldas": Una matriz bidimensional de objetos "Celda" que representa las celdas donde se puede construir algo.
  - " matrizCeldaEnemigos ": Una matriz bidimensional de objetos "CeldaConEnemigos" que representa las celdas donde no se pueden construir pero si tener enemigos.
  - "acreditadorMuertos": Un objeto que implementa la interfaz "OnEnemiesDiedListener" se utiliza para notificar cuando los enemigos mueren.
- Métodos:

- **Constructor** "Mapa": Recibe un lector de mapa, el ancho y alto del mapa, y el objeto oyente para notificar la muerte de los enemigos. Inicializa los atributos y carga las celdas desde el lector de mapa. Y valida el camino sea valido.
- "**agregarCelda**": Agrega una celda al mapa en la posición correspondiente y actualiza las listas de caminos si la celda es “caminable” (o sea si es posible posicionar enemigos terrestres en ella).
- " **obtenerCelda** ": Obtiene la celda donde seria posible construir, de ser posible, sino devuelve null.
- **“obtenerCeldaConEnemigos”:** Obtiene la celda que tiene enemigos, de no existir en la matriz de celda con enemigos, buscara en la de celdas, ya que estas también pueden albergar enemigos.
- "**indexarPasarela**": Busca el índice de una coordenada en el camino terrestre.
- "**moverIndex**": Mueve el índice en el camino terrestre según una cantidad dada.
- "**posicionar**": Intenta posicionar una construcción defensiva en una coordenada dada, de poderse. Esto lo hace mediante un double dispatch, después de validar no sea null. Delegando a la construcción el saber como guardarse en la celda.
- **“posicionarInicio”** posicionar a un enemigo en la pasarela de largada, también ocurriría un double dispatch por medio de posicionarEn, donde Enemigo es el encargado de saber como guardarse en la Celda.
- "**actualizarPosicionEnemigo**": Actualiza la posición de un enemigo desde una coordenada inicial hasta una coordenada final. A excepción de que la coordenada final sea el fin del camino, en ese caso solo se removerá de su posición.
- "**moverEnCaminoAereo**": Mueve un enemigo desde una coordenada inicial a una coordenada final a lo largo del camino aéreo.
- "**moverEnCaminoTerrestre**": Mueve un enemigo desde una coordenada inicial a una cantidad determinada a lo largo del camino terrestre.
- "**accionarEnemigos**": Delega al accionarEnemigos de cada celda que tenga enemigos, quien delega el accionar de cada Enemigo al enemigo.
- "**accionarDefensas**": Análogo a accionarEnemigos, pero con

defensas, se le terminar delegando a la Defensa el accionar.
- "**atacar**": Realiza un ataque a una coordenada por medio de un Atacante, que ahora mismo seria una Defensa, delegando el ataque a la celda en esa posición. Quien tiene un double dispatch, delegándole a la defensa el ataque. La posterior acreditación también es un double dispatch en Jugador. El uso esta ya que se necesita saber información sobre el tipo que no se tiene, pero que el objeto en si, si lo tiene.
- "**atacarPrimeraTorre**": Realiza un ataque que destruye la primera de las defensas provistas por el usuario. En caso de no tener ninguna, no rompe nada.
- "**getInformacionCelda**": Obtiene información sobre una celda específica, como su tipo, posición, vida y defensa.
- "**getInformacionCelda**s": Obtiene información sobre todas las celdas en el mapa.

5.7. Enemigo

La clase Enemigo es una clase abstracta que representa a los enemigos en Algo Defense.
Los enemigos tienen una velocidad que determina su capacidad para moverse por el mapa y atacar a los jugadores.
La clase Enemigo tiene un constructor que recibe la velocidad del enemigo. También tiene métodos para obtener la velocidad y determinar si el enemigo ha sido atacado por una torre defensiva. Además, tiene un método abstracto para determinar si el enemigo ha sido atacado por una trampa (ya que estas reducirán la velocidad del mismo).
El enemigo puede estar vivo o muerto, y se puede acreditar en un sistema de créditos. También proporciona un método abstracto para describir las características del enemigo.
En accionar internamente se delega al método moverse, quien le da instrucciones a Mapa de a donde moverlo, luego si llego al final se efectuara el atacar, el cual efectuara algún ataque al Jugador o a la primer torre.

En el juego nos encontramos con 4 tipos diferentes de enemigos:

- "Hormiga":
  ○Tiene 1 de velocidad
  ○Causa 1 punto de daño al llegar a la meta
  ○Tiene 1 punto de energía
  ○Otorga 1 crédito al jugador cada vez que 1 hormiga es destruida, pero
  cuando se matan más de 10 hormigas, cada nueva hormiga otorga 2

al
jugador.

- "Araña":
  ○Tiene 2 de velocidad
  ○Causa 2 punto de daño al llegar a la meta
  ○Tiene 2 punto de energía
  ○Otorga créditos de forma aleatoria al jugador al ser destruida. Los créditos que otorgan pertenecen al rango 0...10.

- "Topo":
  ○ En sus primeros 5 movimientos tiene 1 de velocidad, luego en los próximos 5 su velocidad es 2, a partir del movimiento 11 su velocidad queda definitivamente en 3.
  ○ Si el número de turno en que llega a la meta es impar, entonces causa 5 puntos de daño, sino sólo 2.
  ○ El topo no camina por la superficie, sino que va enterrado lo que hace que ninguna torre lo pueda atacar

- "Lechuza":
  ○ Tiene 5 de velocidad
  ○ Tiene 5 puntos de energía.
  ○ No causa daño al jugador al llegar a la meta, sino que destruye automáticamente la primera torre construida por el jugador. Si llega una segunda lechuza, destruirá la 2da torre construida y así sucesivamente. Si el jugador no tiene torres entonces la lechuza no produce ningún efecto al llegar a la meta
  ○ Al volar, la lechuza puede moverse por cualquier tipo de parcela, no tiene restricciones. Siempre busca llegar a la meta. Cuando arranca la Lechuza vuela en forma de L hacia la meta (Hace los 2 catetos). PERO cuando la lechuza tiene menos del 50% de vida, cambia drásticamente su forma de volar y pasa a volar en línea recta directamente a la meta (va por la hipotenusa).

Estos tipos de enemigos heredan de la clase base Enemigo principalmente para poder compartir el comportamiento de accionan en el mapa. Moverse y luego si se llego al final atacar. Y se busca poder definir estas acciones internamente sin que un externo tenga que saber como lo hace. Por el lado del EnemigoConVida si se podría hacer alguna composición, pero se prefirió por el momento dejarlo como un tipo abstracto de enemigo del cual enemigos pueden heredar.

5.8. Defensa

La clase Defensa es una clase abstracta que representa las estructuras defensivas en el juego "Algo Defense". Las defensas tienen un estado actual que determina si están en construcción o listas para atacar. Esta idea de estados se implementó de la misma forma que con lo sucedido en Estado Juego. En este caso se tiene un Estado Construyendo y un Estado Construido en donde cada una de estas instancias tendrá un ejecutar diferente.
La clase Defensa tiene un constructor que recibe un estado inicial de la estructura. Tiene un método para finalizar la construcción de la defensa, cambiando su estado actual a "construido". También tiene un método para realizar una acción, que depende del estado actual de la defensa.
Las defensas pueden estar activas o inactivas, según su estado actual. La clase proporciona métodos abstractos para obtener el rango de ataque y el valor de ataque de la defensa. También tiene un método abstracto para atacar a una lista de enemigos.
Las defensas pueden posicionarse en una celda específica del mapa. Además, se puede obtener un descriptor de la defensa que proporciona información sobre su estado y características.

En el juego nos encontramos con 3 tipos diferentes de defensa:

Torre Blanca:
●Coste: 10 créditos.
●Tiempo de Construcción: 1 turno.
●Rango de ataque: 3
●Daño: 1

Torre Plateada:
●Coste: 20 créditos.
●Tiempo de Construcción: 2 turno.
●Rango de ataque: 5
●Daño: 2

Trampa arenosa:
●Sólo puede ser construida sobre una pasarela que no puede ser ni la de llegada ni la meta.
●Ralentiza el avance en un 50% a todos los enemigos que estén en la misma
posición que la trampa arenosa. (No afecta la lechuza, pero sí topo, hormiga y
araña)

- Tiene una vida útil de 3 turnos. Se construye de forma inmediata, es decir al turno siguiente que se la decidió construir ya está operativa por 3 turnos.
- Cuesta 25 créditos.

En resumen, la clase Defensa establece la estructura básica y el comportamiento común de las defensas en el juego. Las subclases de Defensa implementarán los detalles específicos de cada tipo de defensa. Y es por esto que heredan de defensa en vez de usar composición, ya que el comportamiento común de usar un EstadoEstructura, el double dispatch para el ataque a enemigos, etc. Es lógica que se busca quede interna a las defensas y no exponerla.

5.9. Celda

La clase Celda es una clase abstracta que representa una celda en el mapa del juego "Algo Defense". Una celda puede contener habitantes y construcciones defensivas.
Pueden existir tres variantes de esta misma, siendo las mismas Tierra, Rocoso y Pasarela.

Tierra: Es un tipo de celda en donde se pueden colocar torres y por donde pueden circular las lechuzas (tipo de enemigo aéreo). Por lo que hereda de Celda al permitir una construcción. No se usa meramente composición ya que la lógica de que permite guardar y como no debería ser expuesta. Lo mismo para Rocoso y Pasarela.

Rocoso: Es un tipo de celda por donde únicamente pueden circular las unidades aéreas, no es posible construir nada en este tipo de celda. Por lo que hereda directamente de CeldaBase.

Pasarela: Es un tipo de celda por donde circula cualquier enemigo que va avanzando hacia el jugador, además, es la única celda en la que se permite poner trampas para reducir la velocidad de los enemigos. Por lo que hereda de Celda al permitir una construcción.

La celda proporciona métodos para accionar las estructuras defensivas y los enemigos presentes, estos últimos delegados a CeldaBase por composición, utilizando el mapa y la posición correspondiente pasados por parámetros.
También tiene un método para describir la celda, que devuelve un descriptor que contiene información sobre los habitantes, las construcciones defensivas y los enemigos presentes en la celda.
Además, la celda tiene métodos para obtener el daño posible que puede recibir,

eliminar una unidad enemiga de los habitantes, determinar si la defensa puede recibir un ataque aéreo, limpiar las construcciones defensivas de la celda, recibir un ataque de una defensa, retirar enemigos muertos de los habitantes y guardar una defensa en la celda.
Esta clase encapsula la información y el comportamiento de una celda en el juego, incluyendo enemigos, construcciones defensivas y su interacción con el mapa y los jugadores. Aunque su responsabilidad es agrupar a las defensas o enemigos de una posición, y el como accionar es delegado a sus habitantes.

5.10. Excepciones

● CaminoInvalido
Excepción que tira mapa al inicializarse en caso de no instanciarse un camino valido, de 2 o mas pasarelas, continuo.
● CeldaFueraDeRango
Excepción cuando se intenta agregar una celda que se va del rango especificado para el mapa al inicializarse.
● CeldaNoExistia
Excepción que seria tirada en caso de que no se usasen booleanos para manejar el posicionamiento de una construcción Si bien también habría otras razones fuera de esta.
● TamanioInvalido
Excepción tirada por el LectorMapa al detectarse el json no tiene el tamaño especificado.
● EnemigoFaltante
Excepción tirada por el ConvertidorOleada en caso de que falte en el turno la cantidad de Hormigas o Arañas a instanciar. No así si faltan Topos o Lechuzas.
● TurnoFaltante
Excepción tirada por LectorEnemigo cuando se saltea un turno o no está en el orden correcto en el json.
● CambioDeEstadoInvalido
Excepción tirada por Juego en caso de que el cambio de estado que se quiera hacer sea invalido... Empezar un juego empezado o terminar uno terminado.

6. Diagramas de Estados

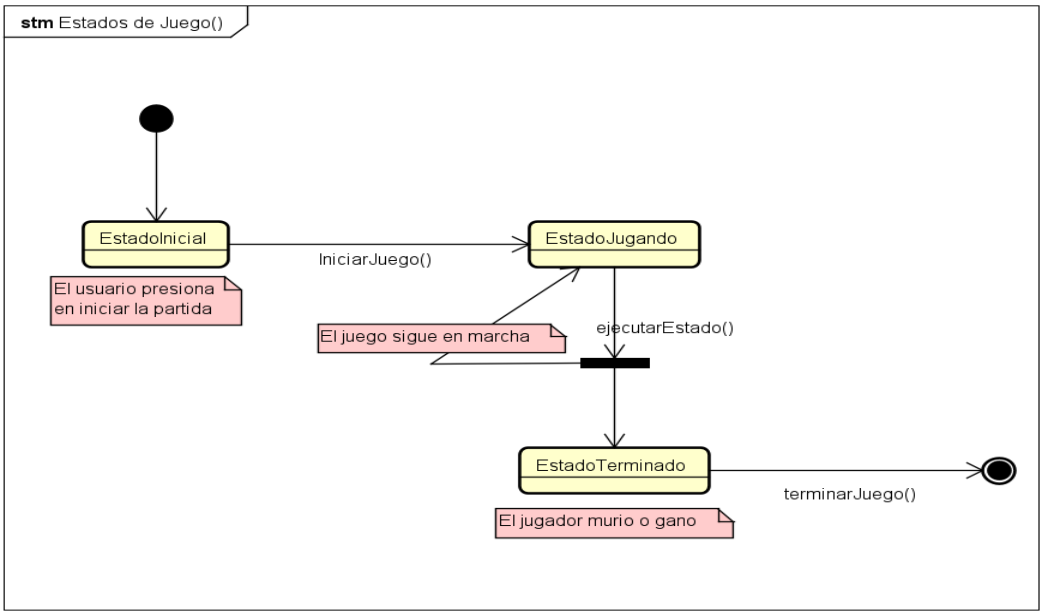


Figura 1 – En el diagrama se muestran los diferentes estados por lo que pasa el juego, estando este en un principio en un “estado inicial” el cual cambia a “estado jugando” cuando el jugador selecciona Iniciar Partida, y finalmente termina en “estado terminado” cuando se capta que el jugador está muerto o que los enemigos ya no son suficientes para poder eliminar al jugador.

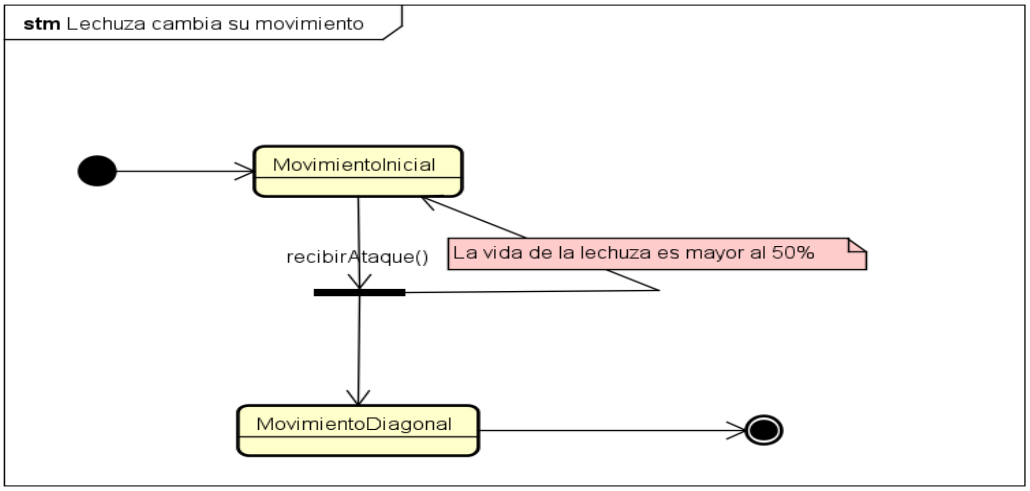


Figura 2 – En el diagrama se muestran los diferentes estados por lo que pasa el movimiento de la lechuza en donde pasa de” movimiento inicial” a” movimiento diagonal” cuando la vida de la lechuza llega a menos de 50%.

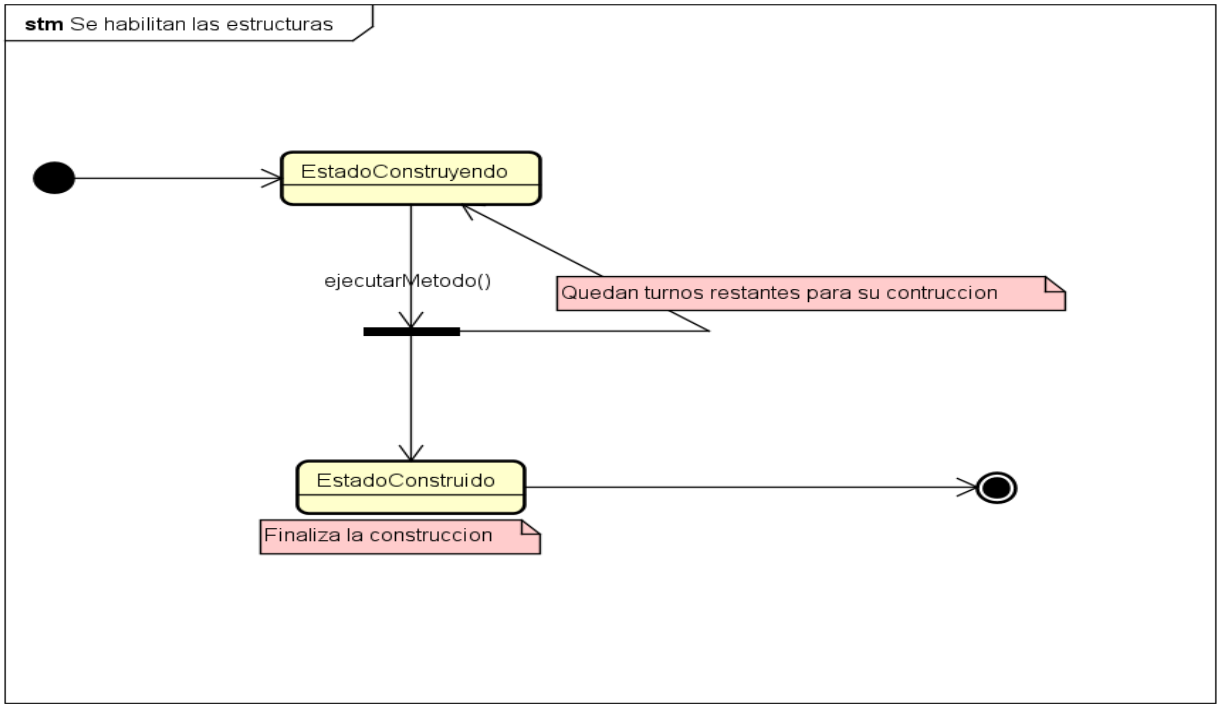


Figura 3 – En el diagrama se muestran los diferentes estados por lo que pasa una construcción. Cada construcción tarda una cierta cantidad de turnos antes de poder completarse. Cuando estos turnos finalizan, la construcción pasa a estar en "estado construido" y es capaz de desempeñar su papel de atacar enemigos.

7. Diagramas de Paquetes

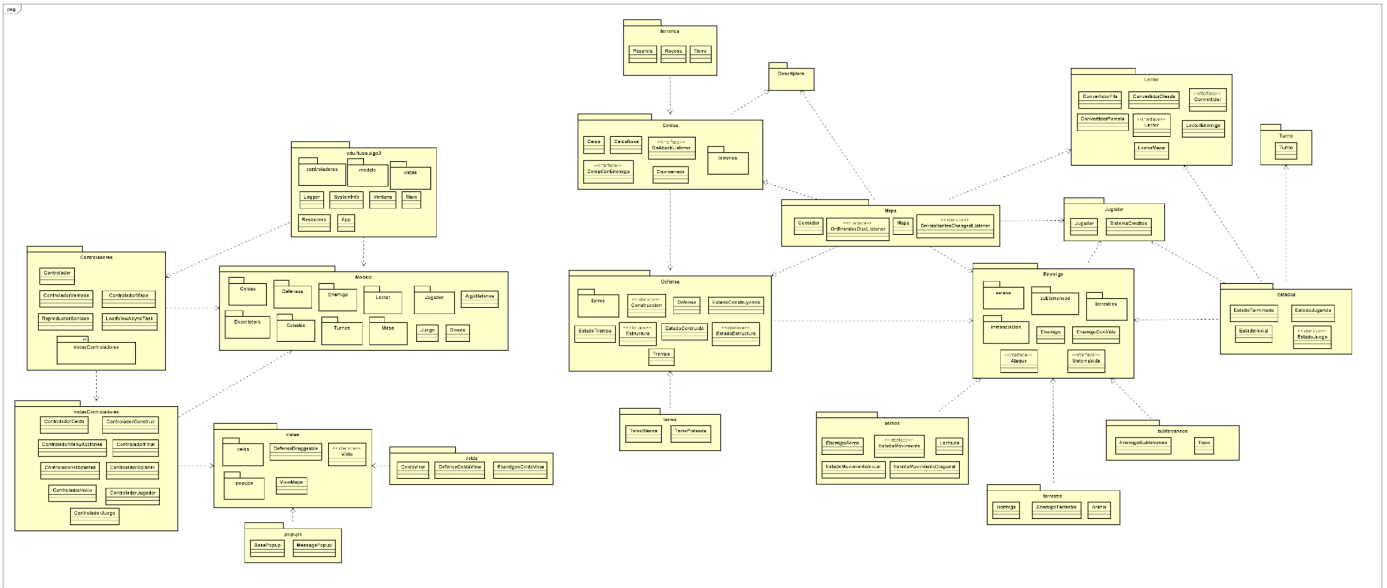


Figura 1 Diagrama de paquetes general

Se observa el uso de una clase intermediaria entre los controladores y los diferentes packages del modelo, los controladores interactuando con AlgoDefense, y así en muchos casos evitando propagar cambios a controladores, además de ser un facade para no exponer directamente a la lógica del juego. También se observa que en el modelo los paquetes realmente están altamente acoplados, esto es porque se decidió no poner

interfaces o abstracciones intermediarias para disminuir estas dependencias. Esto muchas veces generan dependencias circulares que en este diagrama no se pusieron. Entre otras que se omitieron al no ser principales en la funcionalidad.

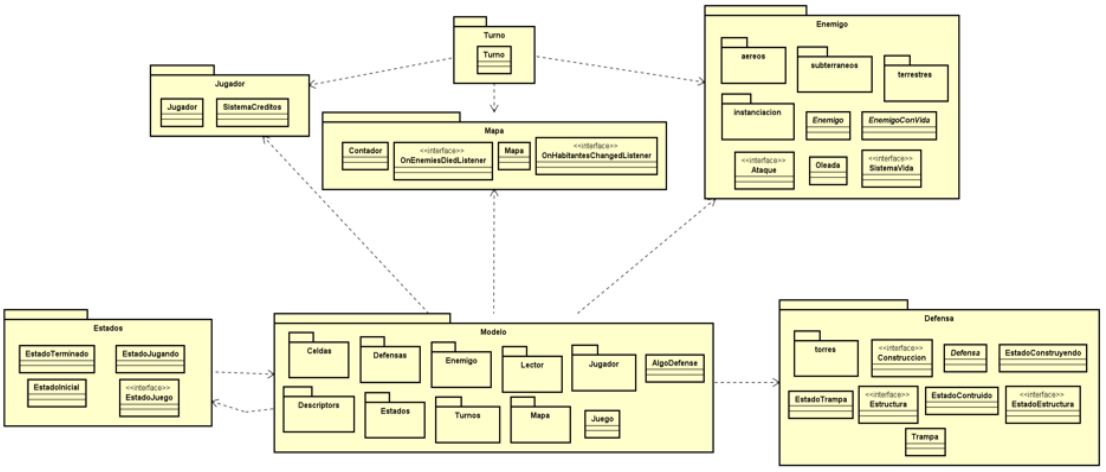
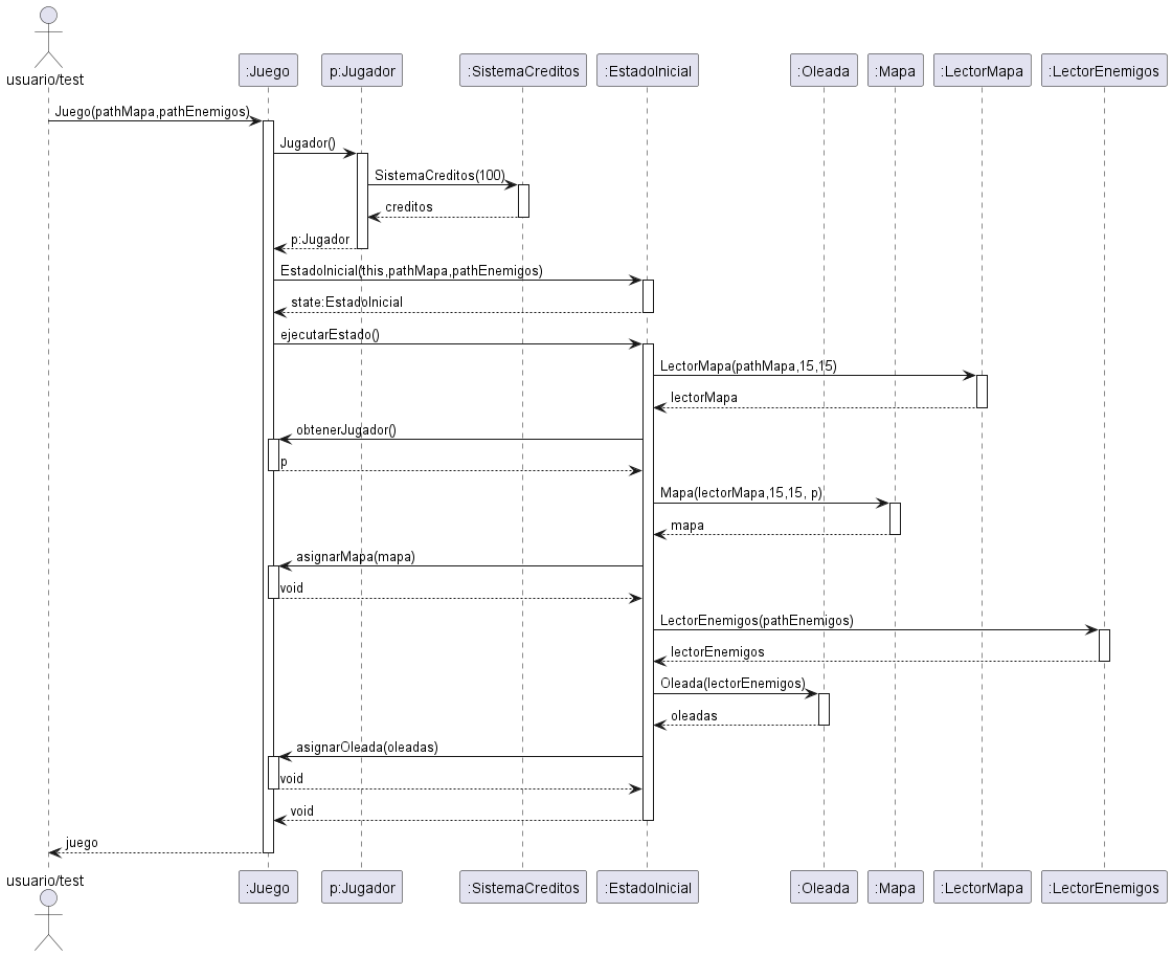


Figura 2 Diagrama de paquetes denotando como Juego perteneciente a modelo y Turno dependen de las clases que ejecutan la lógica en si.
Se observa como turno y modelo comparten la dependencia en Jugador, Mapa y Enemigo, mas especialmente Oleada. Esto ya que ambos utilizan lo que se menciono como Contexto del juego, que seria el Jugador, el Mapa y la Oleada.

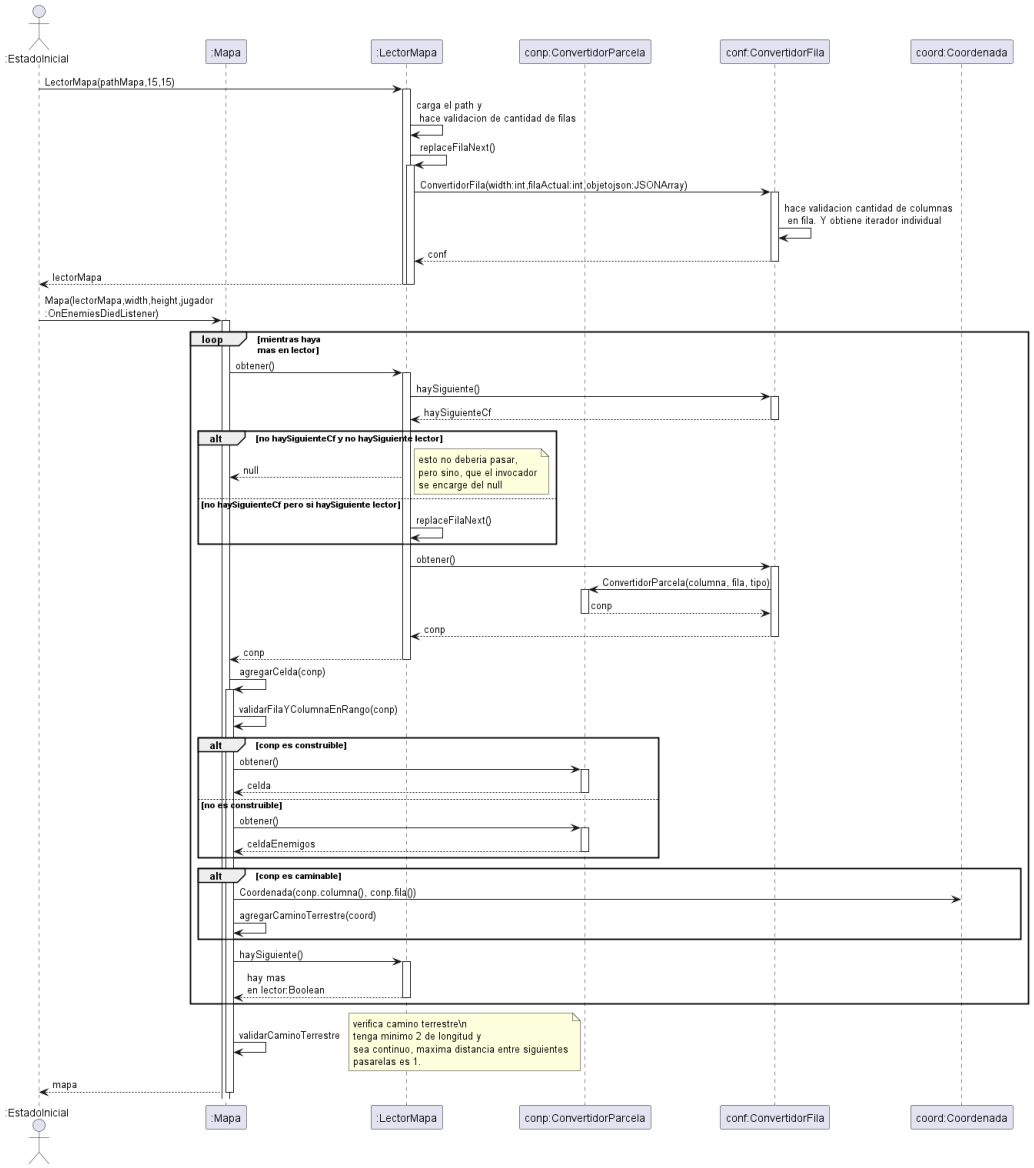
8. Diagramas de secuencia

La primera secuencia importante es ver cómo es que se inicializa juego, el siguiente diagrama describe esto.

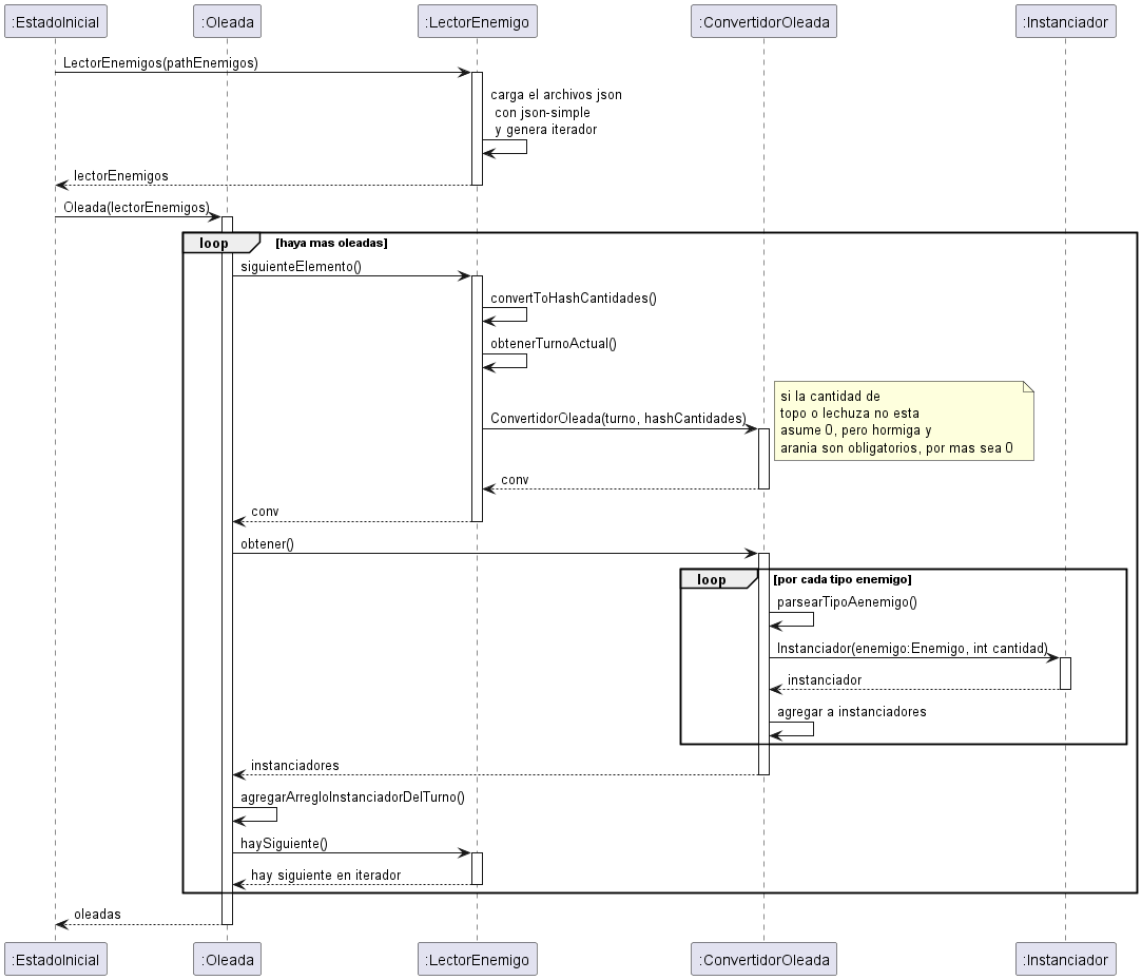


Donde se observa el estado inicial es quien se encarga de inicializar lectores de archivos, creando las clases de lector son las que manejan el formato de este.

A continuación, se observará que pasa internamente en los constructores de Mapa y Oleada y como interactúan con los lectores

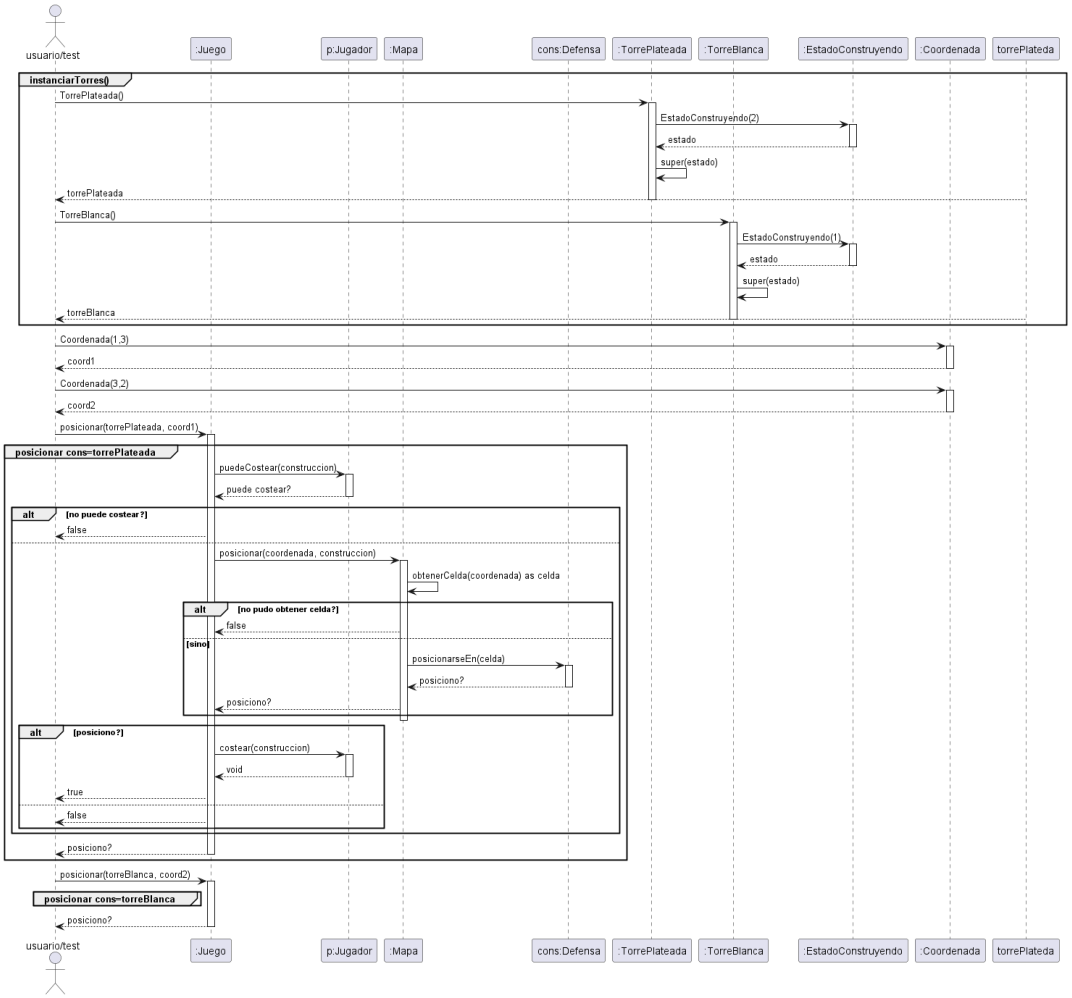


Donde se observa que mapa esencialmente va obteniendo convertidores y en agregar celda se diferencia entre si es posible construir en la celda para separar el contrato de quienes si permiten y quienes no. A la vez se observa que si es caminable se agrega la coordenada a una lista del camino, se asume como se mencionó que el camino es en el orden original, en la validación de no ser así se denotara invalido, es decir si el camino no es un continuo la distancia entre una pasarela y la siguiente es mayor a 1. Además, se verifica que sea mayor a dos pasarelas, es decir un inicio y un fin. No se verifica que sean distintos.



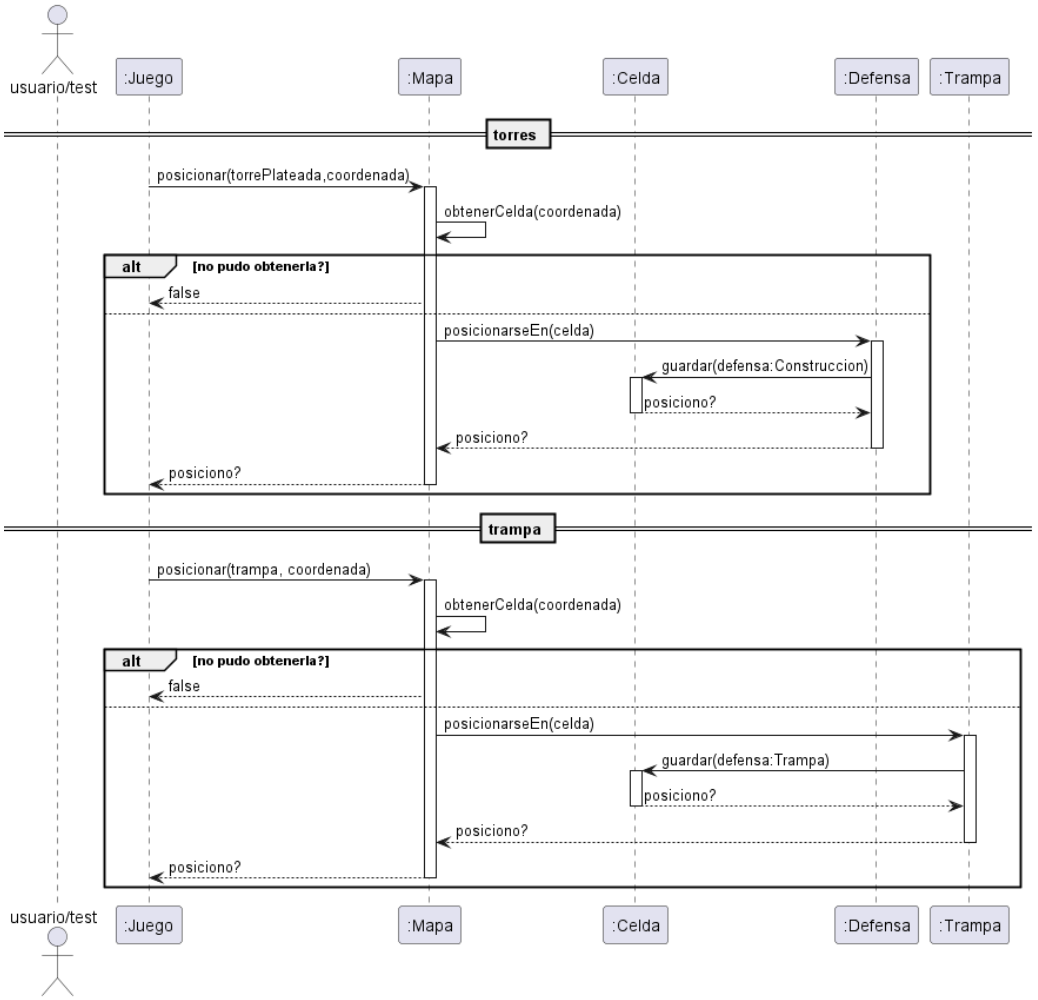
La instanciación de las oleadas es más simple, simplemente se recorre el lector creando a través del convertidor oleada, arreglos de instanciadores para un turno. Se asume el json va a tener todos los turnos, y van a ser en orden. Faltan excepciones verificando eso en el lector.

Posteriormente a la inicialización del juego, se va a denotar en diagramas separados acciones comunes a realizar. Empezando con posicionar una defensa.

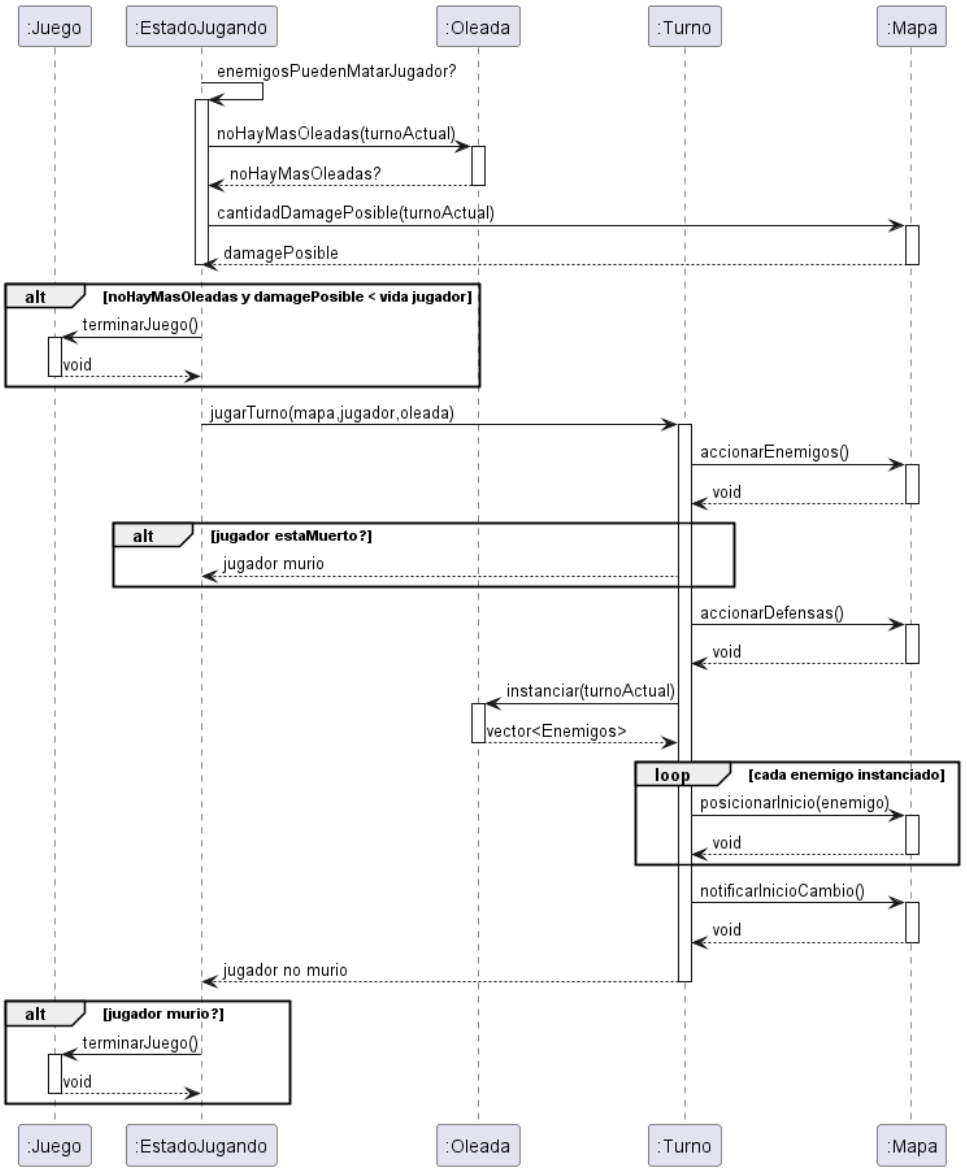


Acá se observa que se rompe medianamente el Tell Don’t Ask en jugador para no asumir que el que interactúe con Juego va a hacer siempre lo correcto, que el jugador pueda costear. En caso de hacer algo incorrecto el juego podría terminar en estado invalido al haber costeado algo el jugador que no debía. A menos que se haga la acción, que agregaría una complejidad extra al modelo. Esto pasaría, aunque se tire una excepción, aunque en realidad nunca debería haber una interacción invalida en este sentido, se prefirió el evadir problemas relacionados a un indeterminismo en ese sentido.

Prosiguiendo, para ser más específicos en mapa al posicionar lo que se hace es un double dispatch, se delega a la construcción el posicionarse en una celda que alberge a estas. Y esta construcción lo que hace es delegarle a la celda, sabiendo información que mapa no sabría, el tipo de defensa que es.

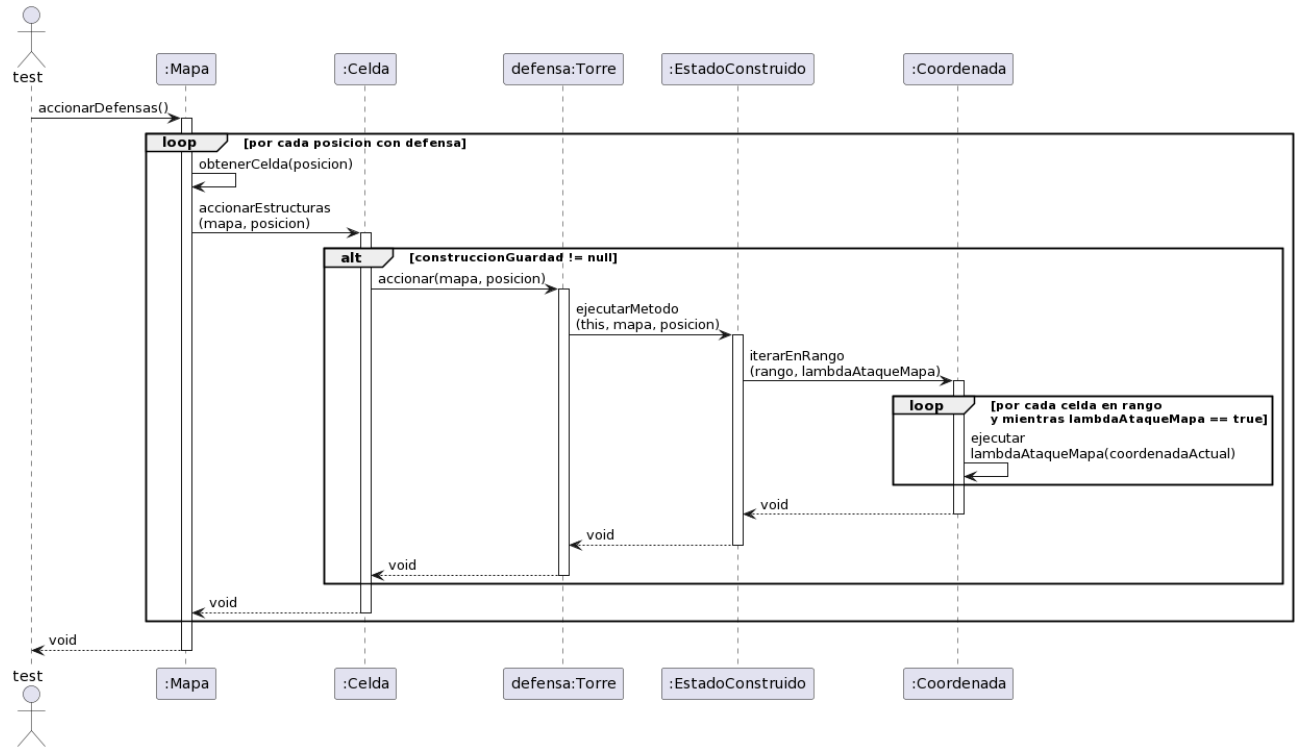


Tras poder construir defensas ahora se podría ver el responsable del pasarTurno de juego, este seria EstadoJugando. Y su pasar turno haría:

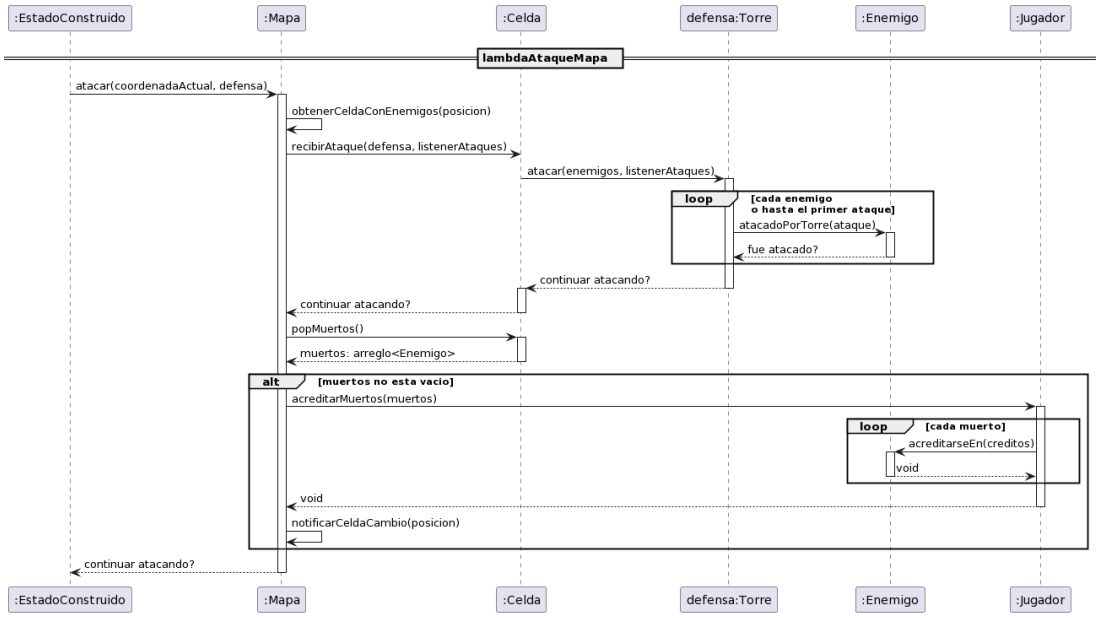


Donde se observa que es EstadoJugando el que se encarga de saber si es que termino el juego, y de haberlo hecho, se lo notifica a Juego para que cambie su estado. La lógica per se de lo que se hace en el turno de juego, se la delega a Turno, quien esencialmente define la secuencia primero atacan enemigos, después defensas y por último se instancian nuevos enemigos.

Por último para entender bien como atacan las defensas vamos a observar



Donde se observa que esencialmente hay una cadena de delegaciones, y que finalmente quien actúa en el mapa es el EstadoConstruido mediante un iterador de rango que define la coordenada. Para esto le pasa una clase anónima que representa un método anónimo, aunque se podría hacer de otras formas, o incluso poner esa lógica en EstadoConstruido, si hizo de esta forma para deja la lógica espacial fuera de lo que seria las defensas. El método anónimo que se le pasa al iterador de Coordenada es



Donde se observa finalmente el double dispatch que existe a la hora de atacar, donde se vuelve a mapa para poder obtener a quien debe recibir el ataque, y luego se le delega a

la defensa el atacar a los Enemigos. Quien con la información de que defensa es, le delega el como recibe el ataque al Enemigo. Hasta que se efectué un ataque, solo ataca a uno.

Luego se quita los muertos de la celda y si hubo muertos entonces se acreditan en el acreditador que seria el jugador, y luego se notifica la celda cambio. Aplicando las partes de diferentes secuencias vistas, se pueden juntar en las siguientes simulaciones de partidas:

Diagrama gana jugador

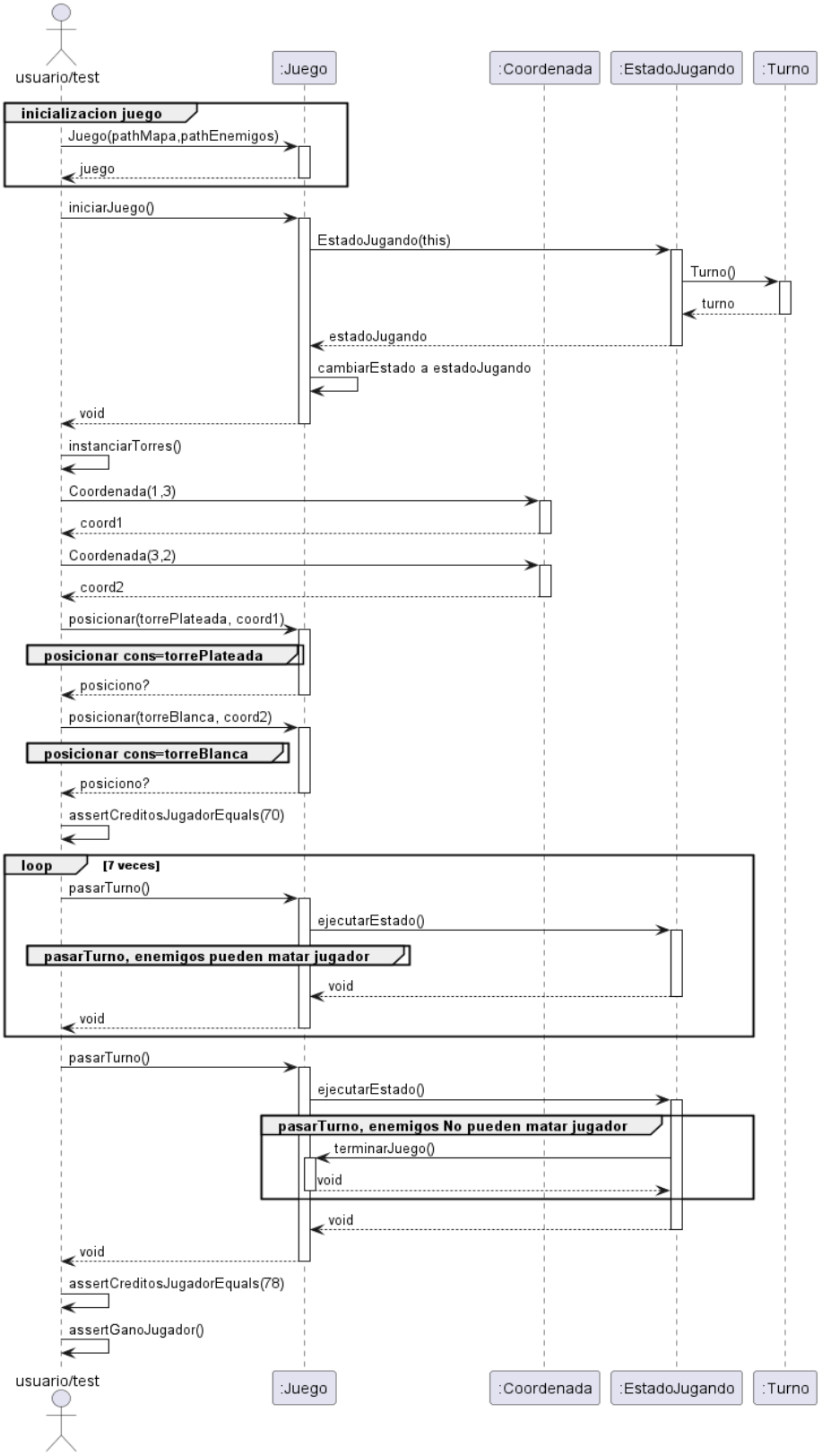


Diagrama pierde jugador