

UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA
Año 2023 - 2^{do} Cuatrimestre

75.07/95.02 - ALGORITMOS Y PROGRAMACIÓN III

TÍTULO: Trabajo Práctico nº2 - "AlgoRoma"

FECHA: 21 de diciembre de 2023

INTEGRANTES:

Abraham, Tomás - #100770
<tabraham@fi.uba.ar>

Bubuli, Pedro - #10342
<pbubuli@fi.uba.ar>

Cumskov, Maximiliano - #101362
<mcumskov@fi.uba.ar>

Ojeda, Galo - #109423
<gojeda@fi.uba.ar>

Porcel, Nicolás - #106244
<nporcel@fi.uba.ar>

Índice

1. Introducción	2
2. Supuestos	3
3. Modelo de dominio	4
4. Diagramas de clase y detalles de implementación	5
4.1. Juego	5
4.2. Mapa, Casillas y Coordenadas	6
4.3. Gestor de Turnos y Jugador	11
4.4. Gladiador	12
4.5. Estado	13
4.6. Seniority	13
4.7. Equipamiento	14
4.8. Log	15
4.9. Parser	17
4.10. Interfaz gráfica	20
5. Diagrama de paquetes	24
6. Excepciones	25
6.1. Excepciones utilizadas durante la creación del mapa	25
6.2. Excepciones utilizadas para el control del programa	27
7. Diagramas de secuencia	28
8. Diagrama de estados	34

1. Introducción

En el presente trabajo práctico se explicará el desarrollo completo de una aplicación llamada 'GLADIATORS'.

La aplicación consta de un juego multijugador que puede ser jugado por un mínimo de 2 jugadores y por un máximo de 6. El juego es un juego de tablero en donde cada jugador tendrá un gladiador y todos comenzaran en la primera posición de un camino que está dentro de un mapa. Los jugadores deberán ir tirando un dado para poder avanzar su gladiador a través del mapa, topándose así con diferentes premios y/o obstáculos almacenados dentro de cada casilla. El primer jugador cuyo gladiador llegue a la casilla final, será el jugador que gane, siempre y cuando su gladiador posea un equipamiento especial que deberá ser reunido a lo largo del juego.

La aplicación fue desarrollada 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. A lo largo del trabajo se explicará detalladamente el diseño de la aplicación y se darán casos concretos de funcionamiento mediante diagramas.

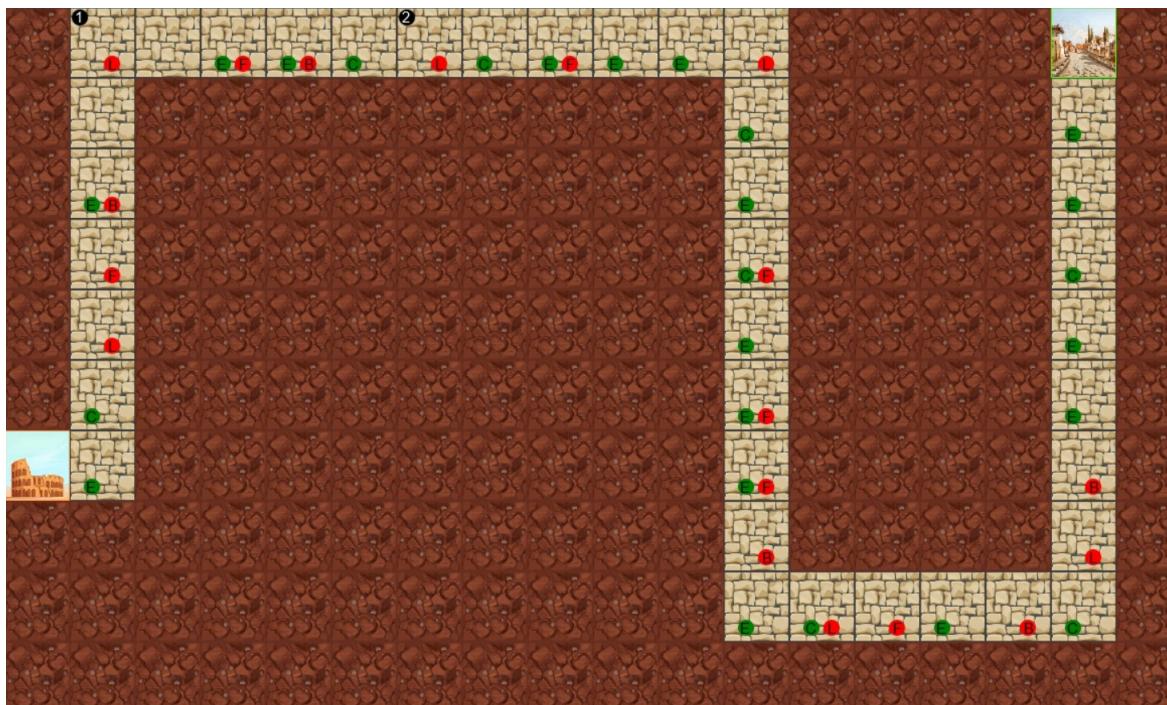


Figura 1: Ejemplo de mapa del juego.

2. Supuestos

En la siguiente sección se exponen los supuestos tomados por el equipo a partir de las situaciones no contempladas en la especificación.

1. En la creación del mapa del juego se asume que lo correcto es recibir un mapa en formato JSON con las siguientes características:

- Las dimensiones del mapa serán positivas mayores a cero.
- La longitud del camino será de al menos tres casillas.
- La primera casilla del mapa será del tipo "Salida".
- La última casilla del mapa será del tipo "Llegada".
- Las casillas intermedias del mapa serán del tipo "Camino".
- Las casillas serán enviadas en orden y formarán un camino continuo (cada casilla será aledaña de la anterior).

Cualquier mapa que no cumpla con las condiciones especificadas será considerado un mapa con formato inválido y no será utilizado para comenzar el juego.

2. En el turno de cada jugador, un Gladiador con estado Sano, primero recuperará energía según su Seniority y luego realizará su movimiento.
3. Cuando evolucione el Seniority de un Gladiador, este recuperará energía en base a su nuevo Seniority.
4. Cuando un gladiador avance más de una casilla en un mismo movimiento se verá afectado sólo por los eventos que estén en la casilla final, si los hubiera, no así por los que estén en las casillas intermedias.
5. Si un gladiador llegara a la casilla final (Pompeya) sin tener el equipamiento que permite ganar el juego (Llave), será trasladado hacia la casilla del medio del mapa. Si en ella se encontrara algún premio u obstáculo el gladiador se verá afectado con ellos.
6. Cuando un Gladiador caiga en una casilla con obstáculo de tipo "Bacanal" la tirada del dado se ejecutará automáticamente, sin requerir acción del jugador.

3. Modelo de dominio

El proyecto se desarrolló en torno a la clase Juego, la cual recibe por inyección de dependencias un mapa, un dado, y un gestor de turnos. El gestor de turnos se encarga de decidir el orden en el que van a jugar los jugadores y suministra al Juego el jugador para el siguiente turno. Al iniciar el turno de un jugador, el mismo deberá a través de la interfaz gráfica tirar el dado para obtener un número entre 1 y 6 y así avanzar su gladiador en el tablero. Los gladiadores tienen energía, equipamiento, seniority y un estado. Un gladiador con energía menor o igual a 0 tendrá un estado de 'sin energía' y no podrá moverse. Al moverse, el gladiador ingresará a un nuevo casillero del mapa. Estos casilleros pueden o no tener:

- Obstáculo: representan un traspie en el camino para el gladiador. Se genera un evento que el Gladiador deberá afrontar, que puede resultar (o no) en la pérdida de energía o cambio de estado del gladiador.
- Premio: tienen un impacto positivo en el gladiador, ya sea con un incremento de energía (en el caso de los comestibles) o con una mejora del equipamiento (en el caso del premio equipamiento).

El juego termina cuando un jugador llega a la casilla final teniendo el equipamiento llave. En ese caso será el ganador. En su defecto, si transcurren 30 turnos de todos los jugadores y ninguno alcanzó el objetivo, el juego finalizará sin un ganador.

4. Diagramas de clase y detalles de implementación

4.1. Juego

La clase Juego es la encargada de gestionar los eventos del juego y de llevar a cabo el transcurso del juego en sí, coordinando a los demás objetos de otras clases para que cumplan sus funciones cuando es debido. En la figura 2 se puede observar el diagrama de la clase con sus atributos y métodos. La clase Juego está completamente implementada contra interfaces, reduciendo así el acoplamiento entre todas las clases y consiguiendo la inversión de dependencias. El Juego ya no depende de las implementaciones específicas de Dado, Jugador, GestorTurnos, o Mapa. De esta misma forma es fácil añadir implementaciones distintas de mapas, jugadores, etc... ya que lo único que deben cumplir es con implementar la interfaz.

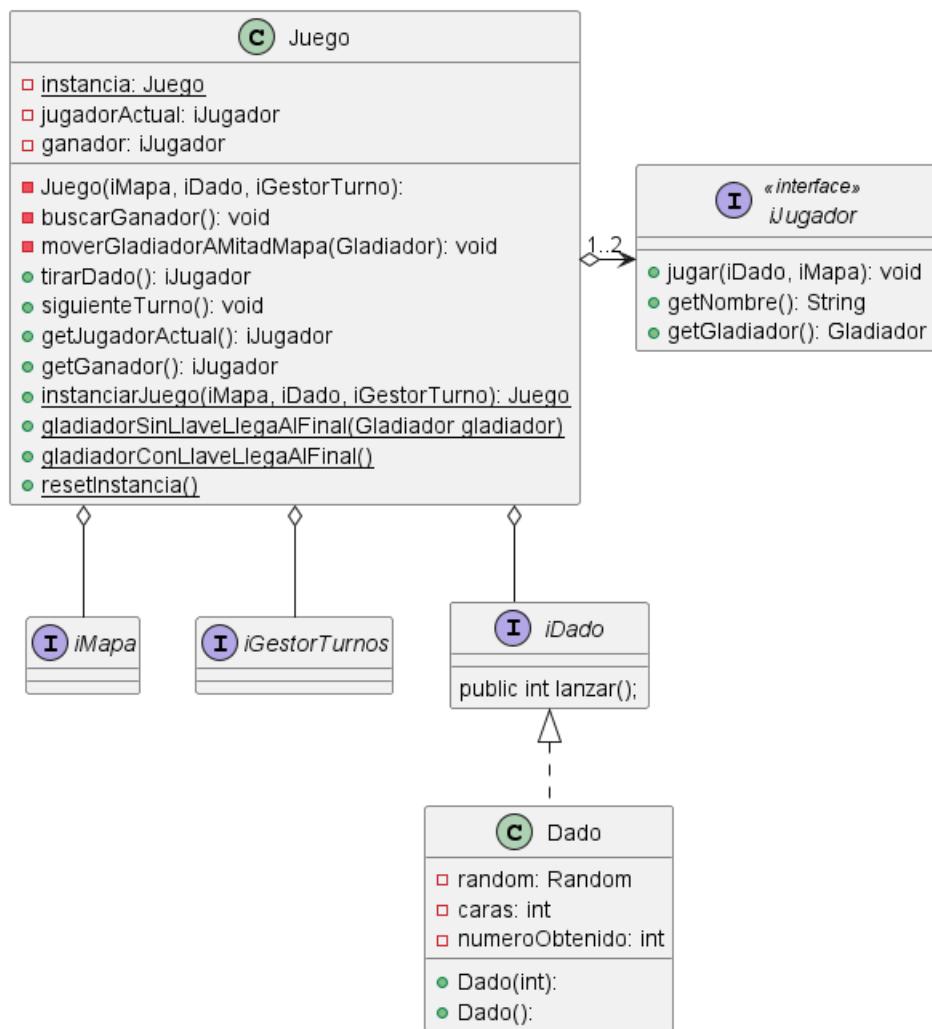


Figura 2: Diagrama de clases de Juego simplificado.

Tal como se aprecia en el diagrama de la figura 2, el Juego tiene dentro de sus atributos implementaciones de un **iMapa**, un **iGestorTurnos**, un **iDado** y 1 o 2 instancias de

implementaciones de iJugador (uno correspondiente al jugador actual y otro correspondiente al jugador ganador si lo hubiera). Los primeros tres son inyectados por parámetro al instanciar el Juego. El Juego será el encargado de ir pidiéndole al gestor de turnos el jugador que debe jugar en cada momento de la partida, y le pasara al jugador el dado para poder 'tirarlo' y el mapa. Cabe aclarar que el mapa que recibe el Juego será un mapa correctamente instanciado según los supuestos ya declarados, y el gestor de turnos estará cargado con los jugadores correspondientes. En las secciones próximas se especificará más acerca de estas clases que comanda el Juego.

Se tomó la decisión de implementar a la clase Juego como un "Singleton". Esto trae como consecuencia que sólo pueda existir una instancia del mismo y esta instancia puede ser accedida desde cualquier parte del programa. Se toma esta decisión para poder dar una solución a la manera de cómo finalizar el juego cuando un jugador gana. La idea de aplicar el patrón Singleton es no tener que enviar por parámetro la instancia de Juego a lo largo de todo el programa, con el único objetivo de que una parte específica de la implementación (que sabe cuando alguien ganó), pueda guardarse la instancia. Siendo más específicos, estamos hablando de los equipables, estos son los que responden a un método polimórficamente e invocan a distintos métodos de la instancia de Juego según sea necesario. Si se ganó, el equipable utiliza el método gladiadorConLlaveLlegaAlFinal() y si no cumple las condiciones para ganar, se llama al método gladiadorSinLlaveLlegaAlFinal().

4.2. Mapa, Casillas y Coordenadas

En la figura 3 se observa un diagrama de clases de la interfaz iMapa y la clase Mapa mostrando sus casillas y su dependencia con Gladiador.

Como se observa el Mapa implementa la interfaz iMapa.

- `ingresarGladiadores()`: para poder ingresar la cantidad de gladiadores que jugarán el juego a la primera casilla del mapa.
- `getLargo()`: utilizada por el controlador de la interfaz para poder graficar el mapa.
- `getAncho()`: idem anterior.
- `getCasillas()`: entrega la lista de casillas, es decir el camino del mapa.
- `moverGladiador()`: utilizada para realizar la acción de mover un gladiador a través del mapa.
- `enviarAMitad()`: es un método invocado por Juego. se utiliza cuando un gladiador llega a la casilla final sin el equipamiento correspondiente y por lo tanto debe volver hacia atrás.

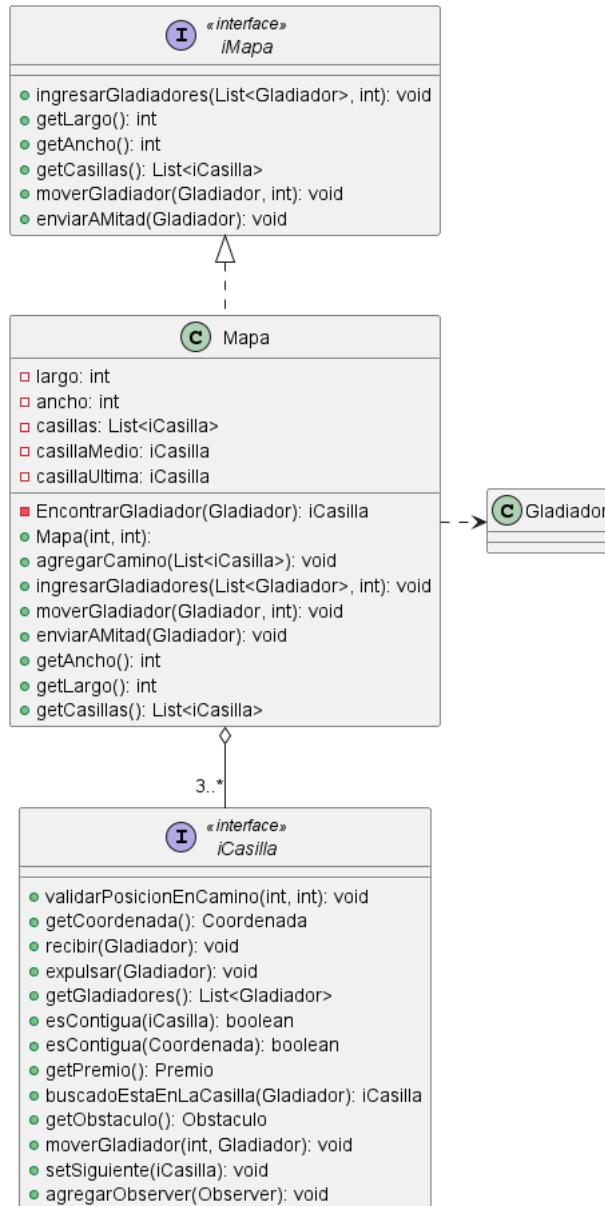


Figura 3: Diagrama de clases de Mapa simplificado.

La clase mapa es la encargada de implementar estas funciones, además de tener un constructor donde se setean las dimensiones del mapa. Básicamente el mapa es el encargado de almacenar la lista de casillas o bien el camino y poder gestionar movimientos de gladiadores entre las casillas. Un Jugador le informa al mapa cual es el gladiador que le pertenece y cuantos pasos debe moverse el mismo. El mapa procede a buscar en su lista de casillas cual contiene al Gladiador buscado y a la que lo contenga le pide que lo mueva, pasándole la información necesaria por parámetro.

Por otro lado el mapa tiene como se mencionó una lista de objetos que implementen la interfaz "iCasilla", que poseen un dato de tipo Coordenada, utilizado para validaciones y para poder graficar las casillas en la posición correcta.

Tal como se puede observar en la imagen 4, la interfaz iCasilla es implementada por las siguientes clases:

1. CasillaInicio: debe ser la primera del camino.
2. CasillaCamino: no puede ser ni la primera ni la última del camino.
3. CasillaFinal: debe ser la última del camino

Las casillas son las que se ocupan de mover al gladiador a lo largo de las mismas. Cada casilla conoce a su siguiente casilla y las 3 clases de casillas responden polimorficamente a los metodos:

1. recibir():
2. expulsar():
3. moverGladiador():

Estos tres metodos estan relacionados entre si ya que son utilizados para el movimiento de los gladiadores entre casillas. El Mapa usa el método moverGladiador() para pedirle al gladiador que avance y le pasa la cantidad de pasos y la casilla en la que se encuentra. Luego, el gladiador se encarga de verificar que no esta lesionado y tiene la energía correspondiente, y si eso se cumple, le pide a la casilla que lo mueva la cantidad de pasos requerida. El método Expulsar() hace que la Casilla elimine de su lista de gladiadores al gladiador que se moverá. Luego las casillas van encadenando llamadas entre si para llegar a la casilla de destino la cual se le indica mediante el método recibir() que debe guardarse en su atributo al gladiador.

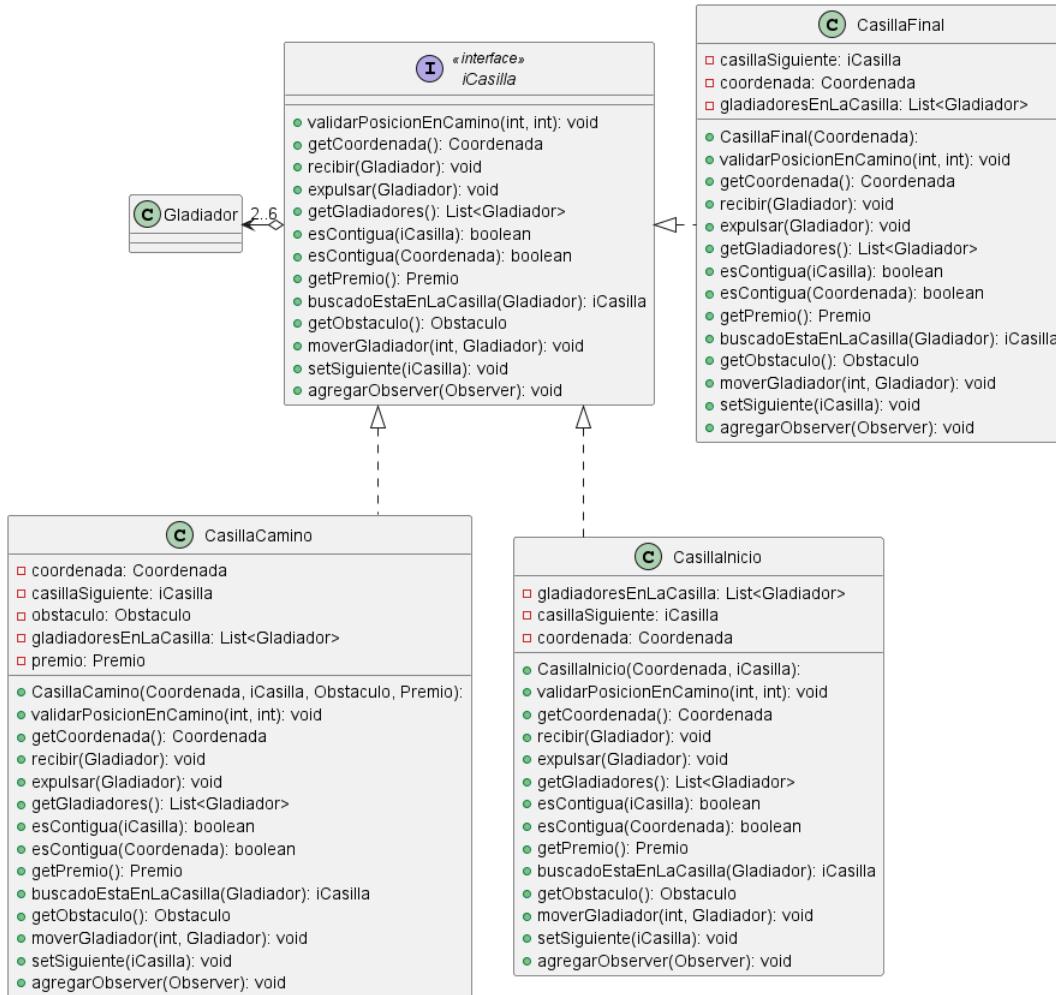


Figura 4: Diagrama de clases de Casillas simplificado.

Por último se muestra en la figura 5 el diagrama de clases de Coordenada y se puede ver que "Casilla Inicio", "Casilla Camino" y "Casilla Final" tienen coordenada. Luego se puede observar también que sólo "Casilla Camino" tiene como atributos un obstáculo y un premio, ya que es un supuesto establecido por el equipo que las casillas iniciales y finales no almacenan un obstáculo o un premio. La clase CasillaCamino, dentro de su implementación del método recibir(), luego de guardarse al gladiador en su atributo, le hace afectarse por el premio y/o obstáculo que contenga.

Tal como se observa en la figura 5, los obstáculos pueden ser del tipo:

- Lesión
- Bacanal
- Fiera Salvaje

y los premios pueden ser

- Equipamiento

- Comestible

Agregando en ambos casos una clase obstáculo u premio nulo para desprenderse de la necesidad de dejar un atributo nulo y poder seguir aplicando el polimorfismo, lo cual facilita la programación porque permite actuar a casillaCamino de la misma manera sea cual sea el objeto, desprendiéndonos de las consultas acerca de qué tipo de objeto es o si hay o no hay un obstáculo/premio en una casilla.



Figura 5: Diagrama de clases de Casillas, coordenadas y eventos.

CasillaCamino no depende de implementaciones específicas de Premios u Obstáculos sino que para su funcionamiento espera la implementación de las interfaces Obstáculo y Premio. De esta forma es sencillo extender el programa creando nuevas clases que implementen una de estas dos interfaces. Para lograr su cometido, los premios y obstáculos responden polimórficamente a los mensajes:

- premiarGladiador():
- obstaculizarGladiador():

ambos métodos reciben al gladiador que deben afectar por parámetro y usan los métodos del gladiador para afectarlo debidamente.

4.3. Gestor de Turnos y Jugador

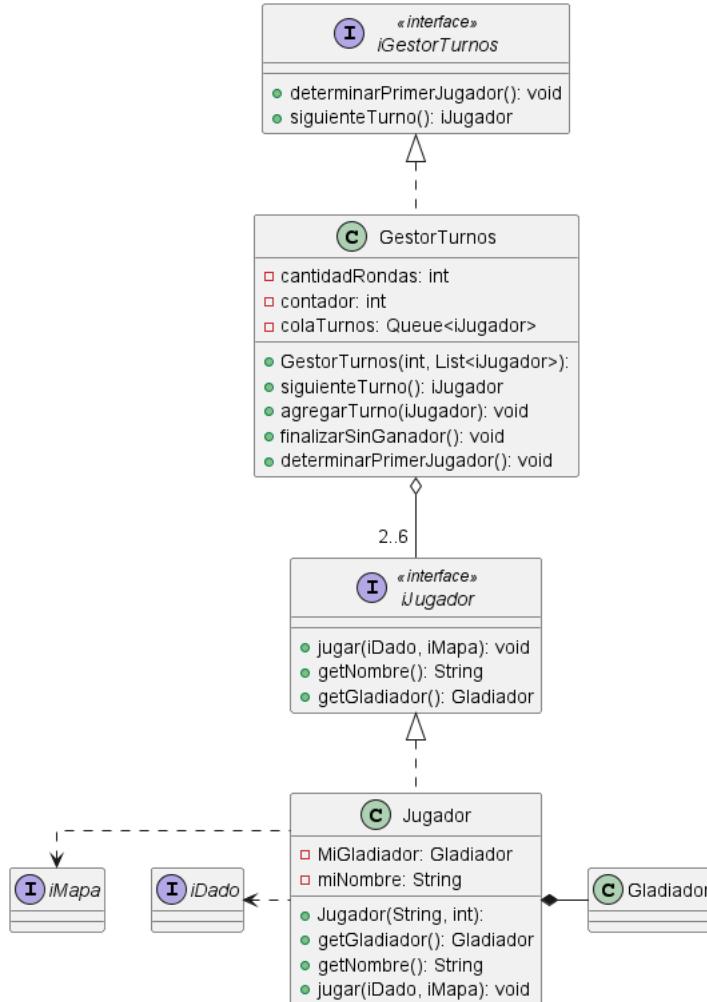


Figura 6: Diagramas de clases `GestorTurnos` y `Jugador` simplificados.

La clase `GestorTurnos` es una implementación de la interfaz `iGestorTurnos`. Para su instanciación, recibe una lista de `iJugador` y un dato de tipo entero que representa la cantidad de turnos a jugarse (por enunciado son 30 turnos por jugador). Durante la ejecución del juego hace uso de una cola para determinar el turno de los jugadores y en el caso de que se superen la cantidad de turnos permitidos, puede lanzar una excepción que es recibida por la clase `Juego`, la cual procede con su implementación para dar la partida como perdida. (ver sección 6 para más información acerca de excepciones).

`Jugador` es una clase que implementa `iJugador`, que se ocupa de instanciar su propio gladiador, almacenar su nombre, ejecutar el dado y pasarle el resultado del dado y su gladiador al mapa para que lo mueva la cantidad de pasos debida. Representa al usuario que estará jugando el juego.

4.4. Gladiador

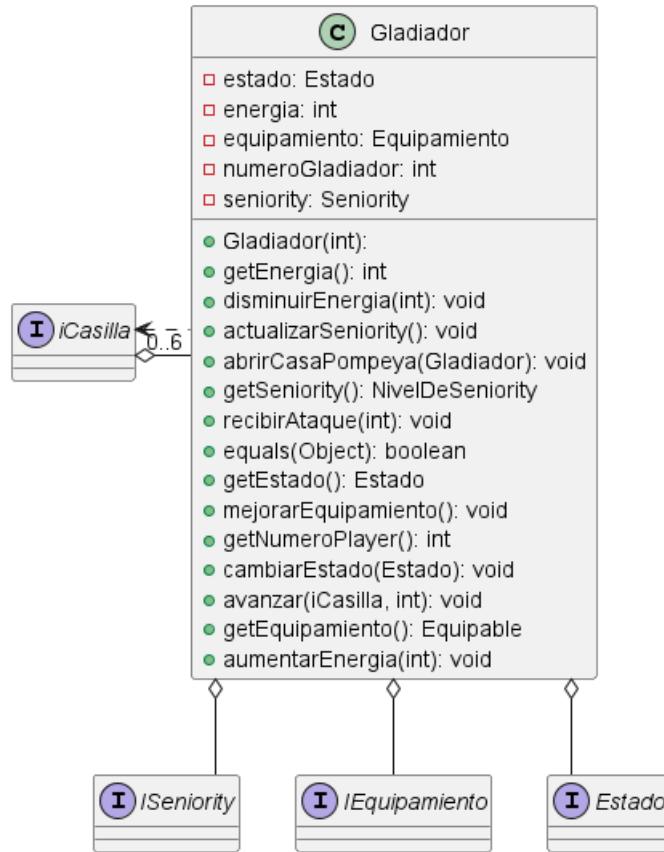


Figura 7: Diagrama de clase Gladiador simplificado.

La clase Gladiador es otra clase muy importante dentro del juego, la mayoría de las Clases ya mencionadas (y próximas a mencionar también), tienen como motivo de existencia afectar al gladiador de una u otra forma. Para la implementación de la clase Gladiador tomamos inspiración del patrón ["State"](#). Lo que más llamó nuestra atención fue el concepto de que la clase adquiriera comportamientos distintos según qué clases tiene guardadas en sus atributos, pero a su vez desconociendo el tipo de clase que tiene almacenada.

El resultado del método avanzar() cambiará dependiendo de qué clase haya guardada en el atributo estado del gladiador, aumentarEnergia() dependerá del seniority y recibirAtaque() y abrirCasaPompeya() dependerán del equipamiento. Al estar el gladiador implementado contra interfaces, este no necesita conocer las clases que tiene almacenadas para poder funcionar. Tampoco depende exclusivamente de ellas, ya que pueden reemplazarse por otras que también implementen las interfaces necesarias y Gladiador seguirá funcionando correctamente.

4.5. Estado

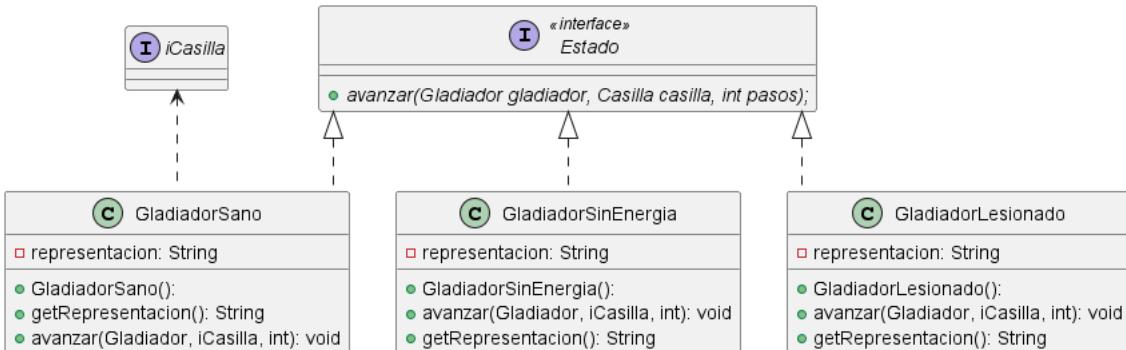


Figura 8: Diagrama de las clases que implementan la interfaz Estado.

Las clases que implementan la interfaz Estado son las encargadas de permitir o no el movimiento del gladiador en su turno. Si el gladiador tiene en su atributo almacenado un Estado de la clase "GladiadorSinEnergia" o "GladiadorLesionado", no podrá moverse y perderá su turno. En cambio, "GladiadorSano" permitirá el movimiento del gladiador y le pedirá a la casilla que contiene al gladiador que lo mueva. De esta manera se respeta la delegación, ya que el Gladiador en sí se desprende de la responsabilidad de saber si es capaz de moverse o no. Cuando recibe la instrucción de moverse, éste la delega hacia su estado, y es el estado quien permite o no la realización del movimiento.

4.6. Seniority

La clase Seniority que implementa iSeniority, se encarga de operar con el NivelDeSeniority para obtener la cantidad de energía correcta que le corresponde ganar al gladiador según su seniority. Novato, SemiSenior y Senior son clases que extienden de NivelDeSeniority y cada una se ocupa de contabilizar cuándo es necesario instanciar el seniority siguiente y cuánta energía se recupera según el nivel.

De este modo la lógica propia del Seniority queda encapsulada dentro de esta clase, Gladiador no debe estar ocupándose de actualizar el nivel de seniority ni tampoco de cuánta energía recuperar en función de qué nivel de Seniority tiene. Esto ayuda a desacoplar el modelo y poder separar responsabilidades de las clases. Del mismo modo también ayuda a lograr un mejor testeo ya que Seniority resulta ser independiente del Gladiador y puede ser testeado sin necesidad de utilizar un objeto de otra clase.

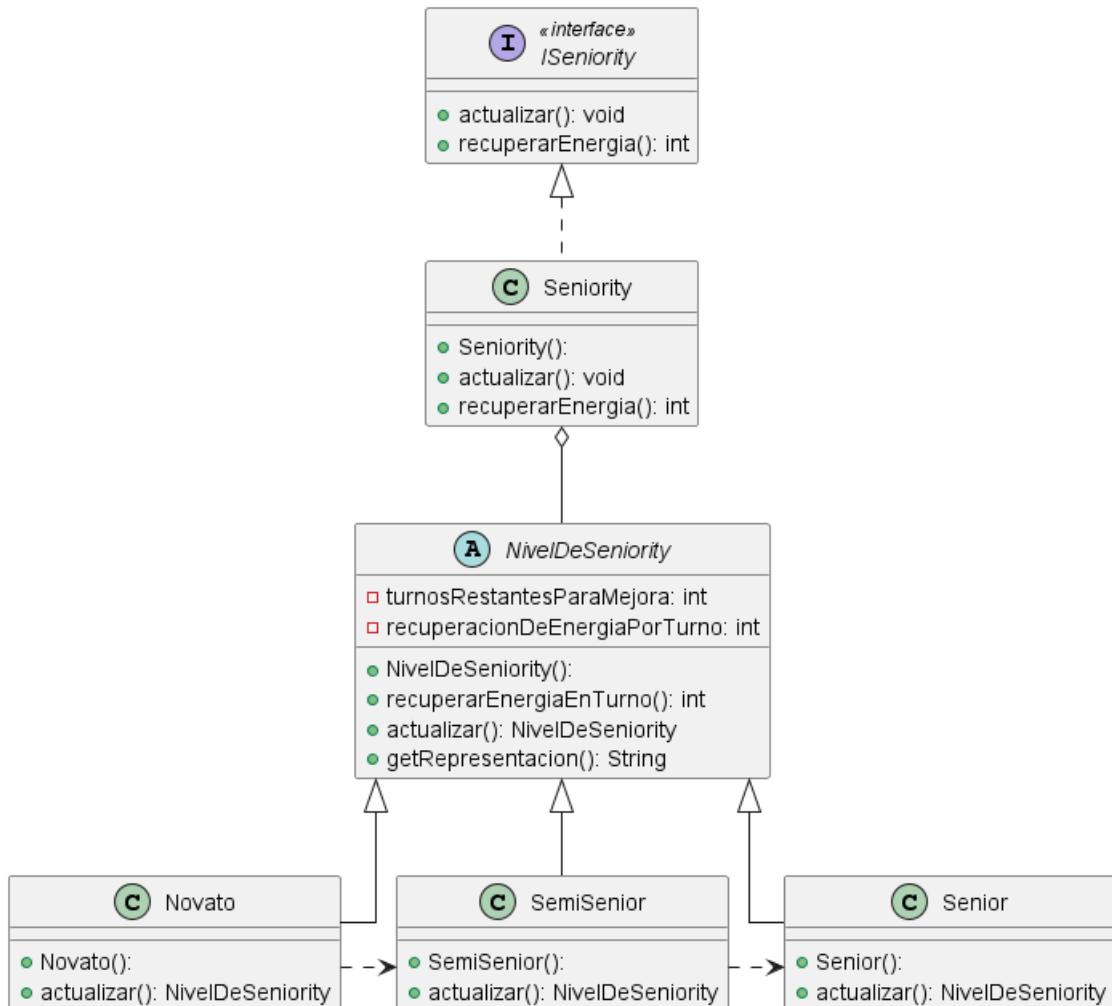


Figura 9: Diagrama de los niveles de seniority de un Gladiador.

4.7. Equipamiento

La clase *Equipamiento* que se puede observar en la figura 10 contiene un objeto de una clase que extiende de *Equipable* que se va a encargar de instanciar a su objeto *Equipable* siguiente cuando sea indicado. *Equipamiento* desconoce qué *Equipable* viene después del otro, ya que es lógica propia de *Equipable*. *Equipamiento* se comunica polimórficamente con los mensajes *mejorar()*, *mitigarDanio()* y *abrirCasaPompeya()*. Cuando un *Equipable* recibe el mensaje *mejorar()*, instancia a su siguiente *Equipable* y lo devuelve para que el equipamiento pueda guardárselo en su parámetro. *mitigarDanio()* recibe un dato de tipo entero que representa daño y le resta a este número su atributo de *danioReducir* para devolver el resultado de daño final. Finalmente, *abrirCasaPompeya()* es un método que provoca que los equipables se comuniquen con la clase *Juego* haciendo uso del Singleton. Si el equipable es una llave, se invoca al método *gladiadorConLlaveLlegaAlFinal()*, cualquier otro equipable invoca al método *gladiadorSinLlaveLlegaAlFinal()*.

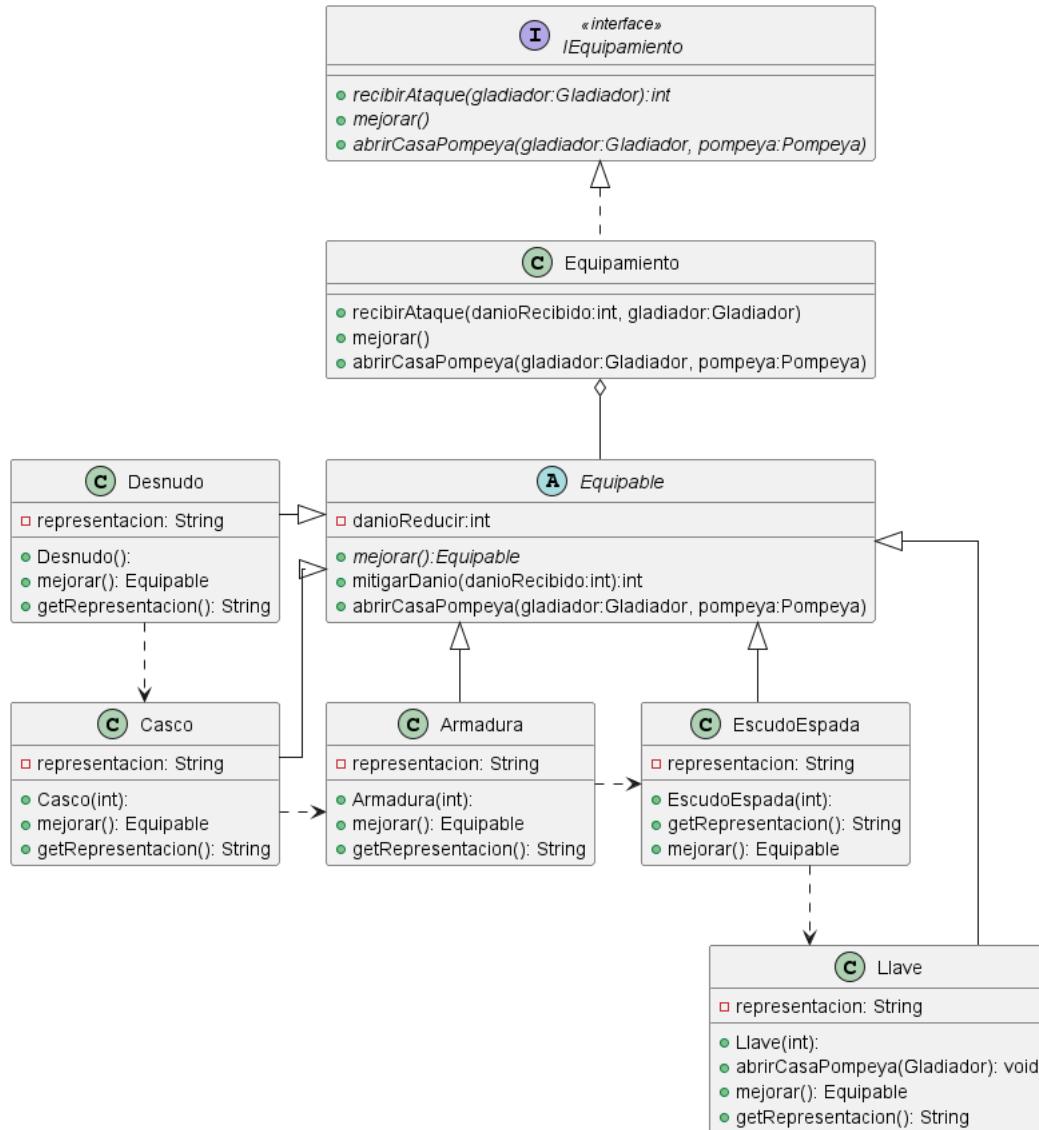


Figura 10: Diagrama de los Equipamientos de un gladiador.

4.8. Log

Log (figura 11) es una clase que se encarga de instanciar un Logger de la librería de Java y construir un formateador para pasarle al logger (LogFormatter es el formateador). Log es un singleton para facilitar el acceso a él desde cualquier clase que necesite notificar un suceso por la terminal. Se uso al logger para ir describiendo lo que va sucediendo en el transcurso de un turno. Log tiene un string de atributo al cual se le van concatenando mensajes nuevos a medida que suceden cosas en el turno y finalmente Juego se encarga de usar el método imprimirMensaje() el cual le envía al Logger el mensaje que se debe imprimir por terminal y luego resetea el string a su estado inicial para que pueda recibir mensajes para un nuevo turno.

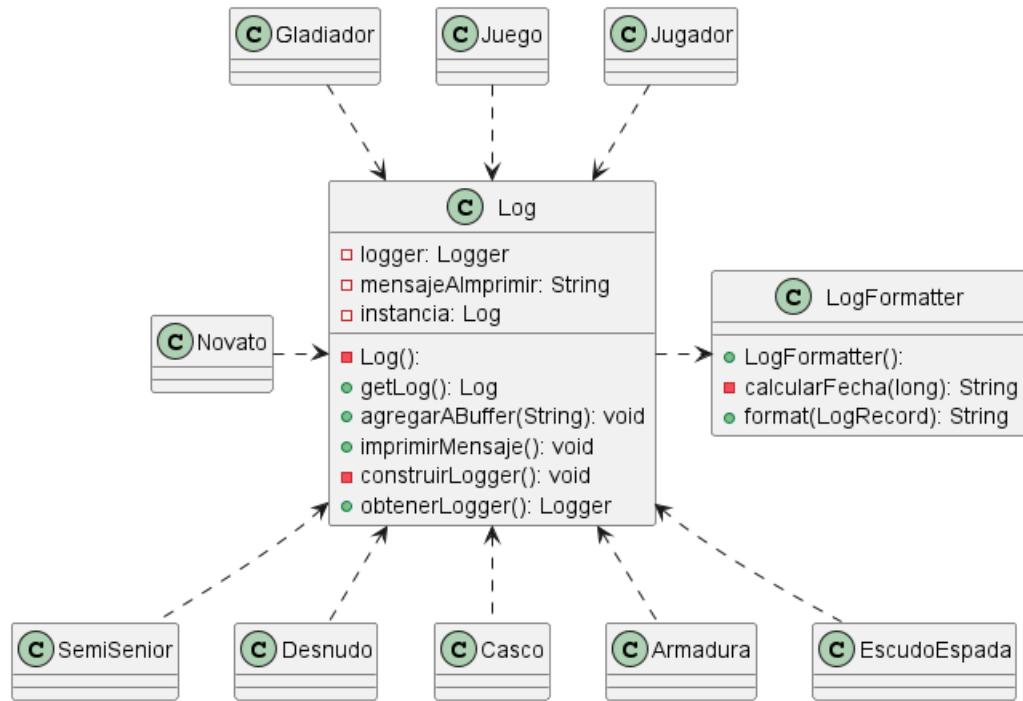


Figura 11: Diagrama de clases Log y LogFormatter.

4.9. Parser

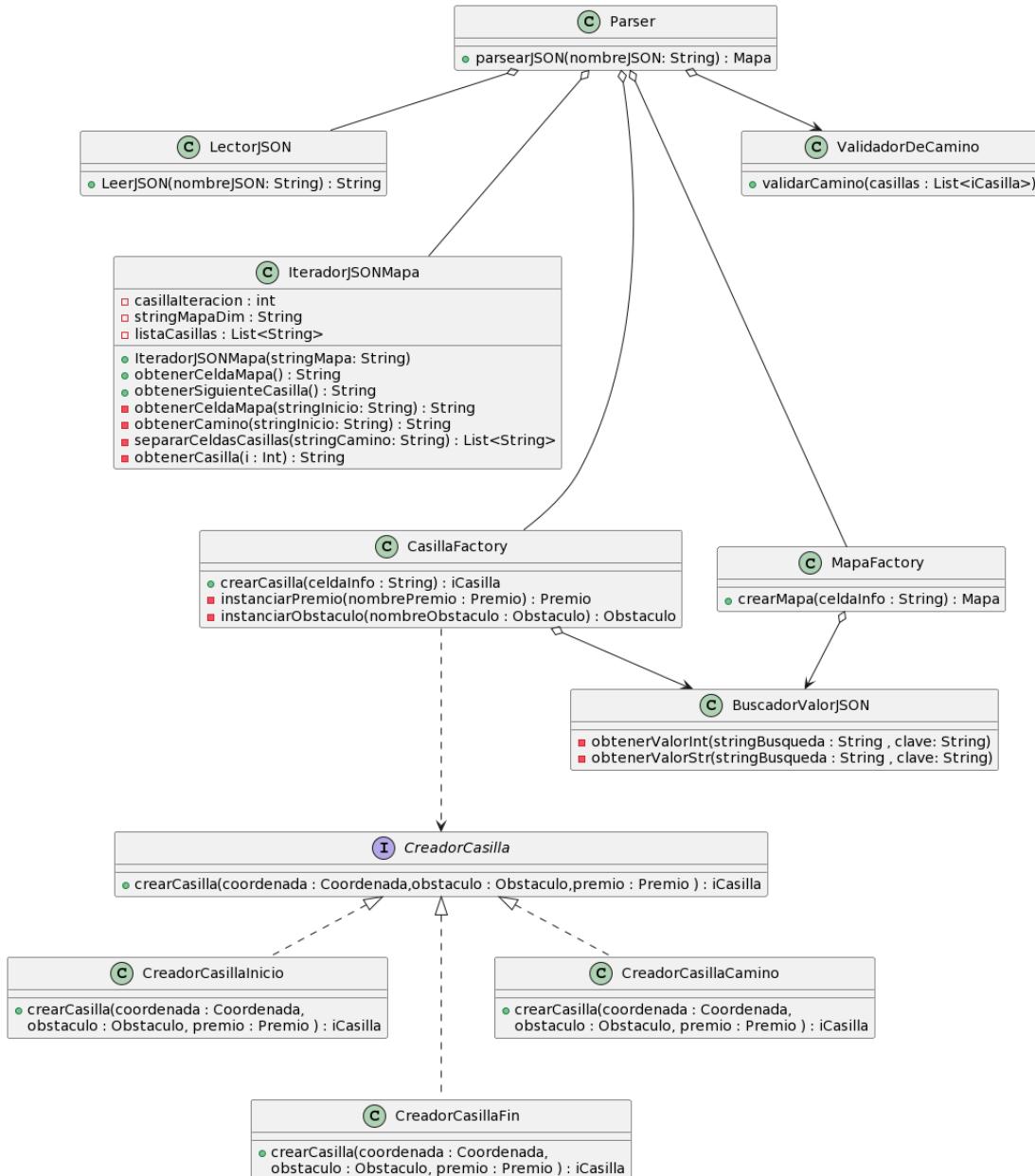


Figura 12: Diagrama de clases general del Parser sin incluir el uso de excepciones.

La clase Parser es la clase que se utiliza para realizar el procesamiento del archivo con formato JSON que especifica el mapa a utilizar. La función de esta clase es recibir una ruta que especifica la ubicación y nombre del archivo JSON, y retorna una instancia de Mapa cargada con sus respectivas dimensiones y con su lista de casillas.

Tal como se puede observar en la figura 12, la clase Parser tiene como atributos diversos objetos de distintas clases, para poder lograr un correcto reparto de responsabilidades a lo largo del procesamiento del JSON y creación del mapa y así cumplir con el principio SOLID de responsabilidad única. En los siguientes ítems se explica la función de cada atributo de Parser y se ahonda sobre la separación de responsabilidades.

1. Parser: objeto principal y contenedor de las demás clases dedicadas a la lectura del JSON y creación del mapa. Su único método, llamado "parsearJSON" recibe como argumento un dato de tipo String con la ruta del archivo a parsear y retorna una instancia de la clase "Mapa" inicializada.
2. LectorJSON: tiene un método "leerJSON", que recibe como parámetro la ruta del archivo JSON. Se encarga de abrir el archivo y cargar todo su contenido en un String. Si puede hacerlo retorna el contenido del archivo a Parser. En caso negativo lanzará una excepción de falla.
3. IteradorJSONMapa: esta clase tiene como responsabilidad iterar sobre el contenido del archivo JSON para poder entregar al Parser todas la información contenida dentro de las celdas del archivo. Esta clase contiene los siguientes métodos públicos:
 - Constructor: cuando una instancia de esta clase es creada, se pasa como argumento el contenido de JSON que entregó el Lector de JSON.
 - "obtenerCeldaMapa()": entrega una cadena de texto con la porción del JSON correspondiente a la celda con la información de las dimensiones del mapa.
 - "obtenerSiguienteCasilla()": en su primer llamado devuelve la celda del JSON que tiene la información de la primera casilla. En su segundo llamado entrega la información de la segunda casilla, e irá actualizando su contador interno hasta que se acaben las casillas para retornar.
4. "BuscadorValorJSON": es una clase auxiliar creada con el objetivo de tener un objeto que pueda buscar el valor de una clave dentro de un string en formato JSON, ya sea un valor de tipo string o un valor de tipo entero. Este objeto puede lanzar una excepción de clave inexistente en caso de que se pida una clave que no está dentro de la cadena recibida.
5. "MapaFactory": clase encargada de instanciar el mapa. Su método público "crearMapa" recibe como argumento una cadena de texto con la información del mapa. Conociendo lo que esa cadena debe contener, busca esa información en la cadena a través de su atributo que es de la clase "BuscadorValorJSON", explicado anteriormente. Por último retorna el mapa instanciado o una excepción por falla (formato del mapa inválido).

6. "CasillaFactory": su funcionamiento es similar al de MapaFactory. Esta tiene su método público "crearCasilla" que recibe una cadena de texto con la información de la casilla y retorna un objeto que suscribe a la interfaz "iCasilla". En una primera instancia se extraen de la información recibida las claves que correspondan para una casilla a través del objeto de tipo "BuscadorValorJSON", que también tiene como atributo. En el caso de esta clase se buscó implementar el patrón de diseño "[Factory Method](#)" ya que éste resuelve la necesidad de poder crear objetos de cierta interfaz desde una clase madre pero permitiendo que las clases hijas puedan crear objetos de distintas clases, siempre que suscriban a la misma interfaz. De este modo "CasillaFactory" tiene acceso a las clases que suscriben a la interfaz "CreadorCasilla". CasillaFactory es capaz de detectar que un premio, obstáculo o tipo de casilla es inválido y lanzar una excepción correspondiente.

7. "CreadorCasilla": es la interfaz a la que suscriben los creadores de una clase concreta de casilla. Estos son:

- "CreadorCasillaInicio"
- "CreadorCasillaCamino"
- "CreadorCasillaFinal"

Todas estas clases tienen su método "crearCasilla" que deben implementar por estar suscritas. Este método recibe como parámetro una instancia de Coordenada, una instancia de Premio y una instancia de Obstáculo. Luego cada tipo de creador hará lo que tenga que hacer con esa información para poder instanciar el objeto iCasilla correspondiente.

8. ValidadorDeCamino: es una clase creada con el objeto de lograr una validación del camino creado. Este objeto recorre el camino que se creó y valida:

- Que la primera casilla sea de tipo Salida.
- Que la última casilla sea de tipo Llegada.
- Que las casillas intermedias sean de tipo Camino.
- Que el camino sea continuo.
- Que ninguna casilla esté por fuera de las dimensiones del mapa.

Puede lanzar una excepción si detecta que alguna de estas condiciones no fue cumplida. En caso de que todas las condiciones hayan sido debidamente verificadas, se podrá finalmente instanciar el mapa creado en condiciones de ser utilizado.

En la sección 7 se muestran algunos ejemplos de cómo funciona el Parser y demás clases mencionadas a lo largo de esta sección.

4.10. Interfaz gráfica

Haciendo uso de JavaFX se hizo una interfaz grafica guiandose por MVC. A continuacion dejamos unos diagramas ilustrando el funcionamiento de la interfaz.

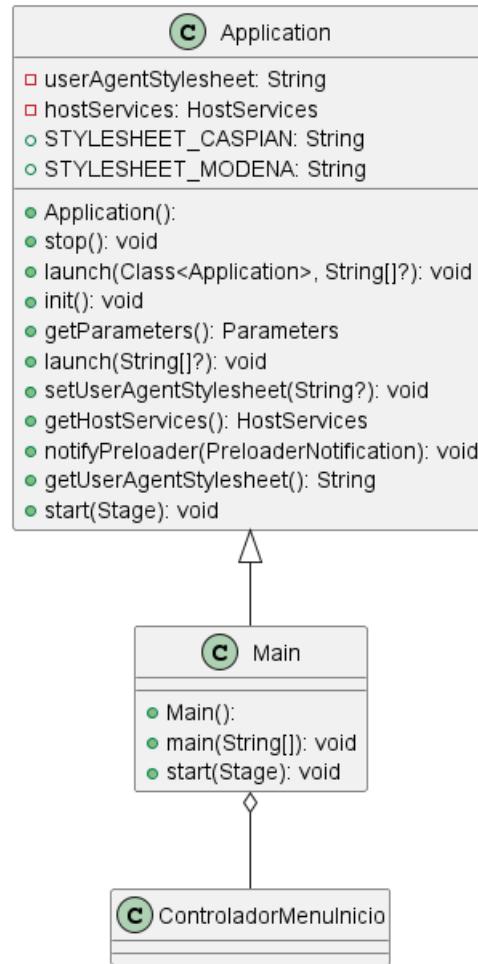
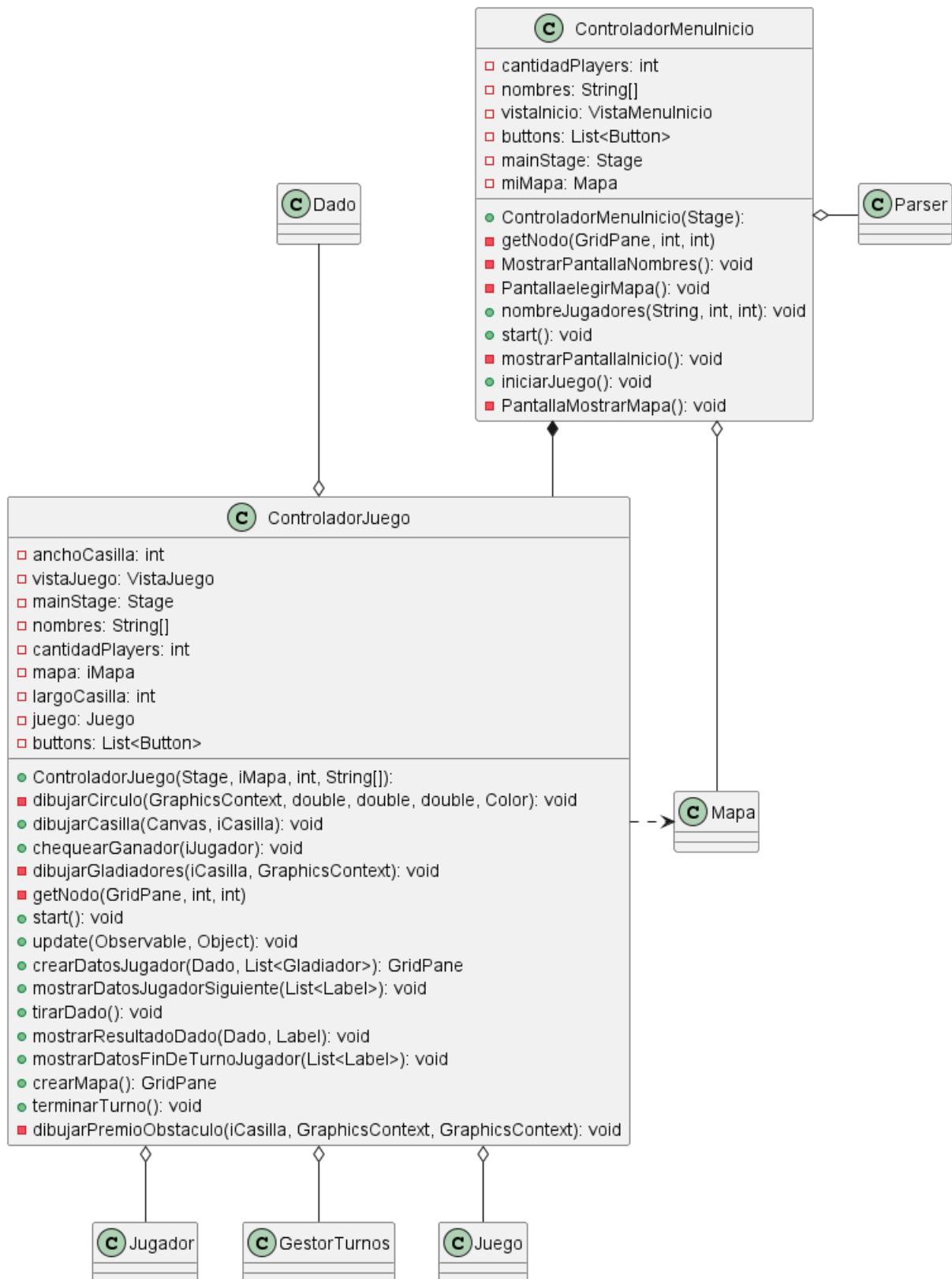


Figura 13: Diagrama del Main y el instanciamiento del Controlador.

El juego comienza a ejecutarse desde la clase Main, la misma hereda de Java Application. En el método start, es instanciado un ControladorMenuInicio especificándole el Stage que utilizará para realizar los diferentes renders de las vistas.



Figura 14: Diagrama de los Controladores y las vistas.

**Figura 15:** Diagrama de los Controladores y el modelo.

ControladorMenuInicio se encarga de instanciar los botones que se mostrarán en la VistaMenuInicio. Estos botones al ser clickeados invocan métodos del ControladorMenuInicio, que se encargan de pasar las escenas de la VistaMenuInicio enviándole por parámetro también los datos necesarios para poder ejecutarla. A través de la vista, el usuario elige la cantidad de Jugadores. Si es un numero valido se habilitara el botón para pasar de escena. Luego el usuario escribe los nombres para los jugadores, y la vista le pasa esta información al controlador que luego hará uso de ella cuando sea necesario instanciar la clase Jugador.

ControladorMenuInicio instancia un Parser y a través de la vista recibe un JSON con el cual se obtendrá una instancia de un Mapa. Si el JSON es inválido no se instanciara un mapa y el botón para pasar de escena no se habilitará. Una vez que fue ingresado un Mapa valido se puede pasar a la última escena de la VistaMenuInicio. El controlador crea un "GridPane" para representar de forma simple el mapa seleccionado y se lo pasa a la vista, para que el usuario pueda ver si el mapa seleccionado es de su agrado o no. Si no decide volver a la escena anterior, puede tocar el botón para iniciar el Juego.

Después de usar el botón para iniciar el juego, ControladorMenuInicio instancia un ControladorJuego pasándole el Mapa instanciado, los nombres de los Jugadores y la cantidad de Jugadores por parámetro. ControladorJuego inicia instanciando botones para la vista e instanciando objetos de la clase Jugador con los nombres correspondientes, un gestor de turnos y un Juego. Para instanciar la VistaJuego, el controlador le pasa los botones y dos GridPanes. Un GridPane es para poder representar el mapa, y el otro se usa para poder representar la información del Jugador del turno actual. Estos GridPanes son construidos por el ControladorJuego y hacen uso del método getRepresentacion() que tienen las clases del modelo que necesitan ser representadas de una forma específica. De esta forma cada clase del modelo tiene en su atributo una cadena de caracteres o un carácter que lo representa y el controlador obtiene esta información para representarlo con eso. De esta forma logramos evitar tener que verificar el tipo de objeto que es y evitamos también tener que hacer una gran lista de if's que indiquen como representar visualmente a cada tipo, violando por completo el polimorfismo.

Para lograr que la representación del mapa y la información del jugador se actualicen a medida que hay cambios, se hace uso de la clase Observable de Java. Juego, Gladiador, Dado y Casilla extienden de la clase Observable y cuando se registran cambios que deben ser representados en la interfaz, el Observer llama a los métodos del ControladorJuego para que re-dibujen el cambio en la Vista.

Cuando el Juego notifica a su Observer que hay un ganador, el controladorJuego instancia el ControladorPantallaFinal y le dice que debe mostrar la VistaGanador. En cambio si el controlador catchea la excepcion SinGanadorException, instancia un ControladorPantallaFinal y le indica que debe mostrar la VistaPartidaPerdida.

5. Diagrama de paquetes

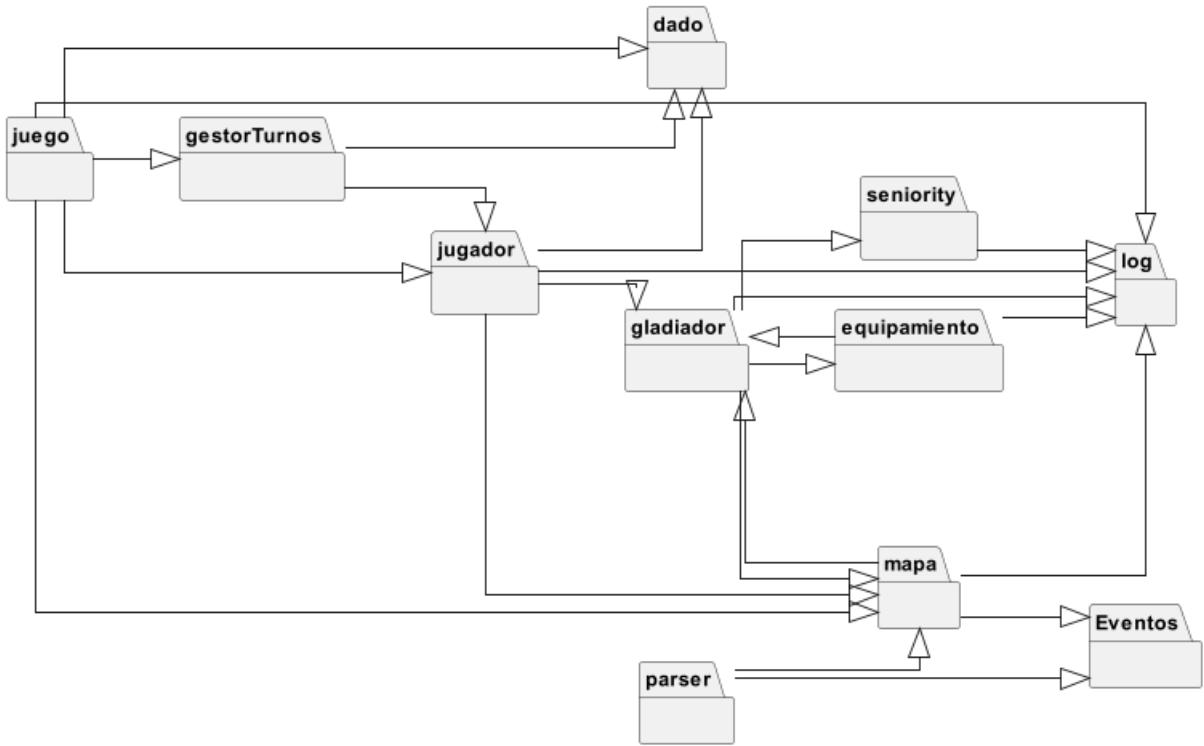


Figura 16: Diagrama general de todos los paquetes

6. Excepciones

A lo largo del desarrollo de la aplicación fue crucial tomarse el trabajo de buscar las situaciones que pueden interrumpir el flujo normal de ejecución del programa. Para abordar estas eventualidades, se recurre al uso de excepciones. Las excepciones se utilizan para no dejar que estos eventos interrumpan abruptamente la ejecución del programa, ya que ofrecen una manera estructurada de manejar estas situaciones y permiten tomar medidas adecuadas.

A continuación, se enumerarán y desarrollarán las excepciones utilizadas a lo largo del programa, explicando ante qué evento pueden ser arrojadas.

6.1. Excepciones utilizadas durante la creación del mapa

En la creación del mapa es donde se vio la necesidad de crear una mayor cantidad de excepciones, ya que es el único lugar del programa en donde se necesita recibir un archivo y validarla en su totalidad. Es necesario que éste cumpla con ciertos criterios desde su formato hasta su contenido, metiéndose con estructuras propias del modelo y demás. Las excepciones creadas y utilizadas son las siguientes:

- "NoSePudoAbrirArchivoException": esta excepción se utiliza cuando el método que abre el archivo con el mapa no es capaz de abrir el mismo, ya sea por mala indicación de la ruta o por otros motivos.
- "FormatoInvalidoMapaException": se lanza en caso de que el archivo que contiene la información del mapa no tenga la información ni la estructura necesaria.
- "ClaveInexistenteException": se utiliza en caso de haber buscado cierta clave correspondiente al mapa o a casillas en una celda del JSON y no haberla encontrado.
- "MapaDimensionesInconsistentesException": se lanza en caso de que el mapa ingresado tenga dimensiones menores o iguales que cero.
- "CasillaTipoSalidaMalPosicionadaException": es instanciada cuando la primera casilla del camino no es del tipo "Salida", ya que es pre-condición que debe el camino ordenado.
- "CasillaTipoLlegadaMalPosicionadaException": se utiliza cuando la última casilla del camino no es del tipo "Llegada", es otra pre-condición asumida.
- "CasillaTipoCaminoMalPosicionadaException": se utiliza cuando una casilla intermedia del camino no es de tipo "Camino".
- "CaminoDiscontinuoException": se utiliza cuando se detecta que un mapa entregado no es continuo, es decir que una o más de sus casillas contiguas no son aledañas.
- "CasillaFueraDeMapaException": es arrojada cuando una casilla del camino excede las dimensiones del mapa.

- "ObstaculoInvalidOperationException": se utiliza cuando al buscar el valor de un obstáculo se encuentra algo que no cumple con lo que se espera recibir.
- "PremioInvalidOperationException": se lanza cuando al buscar el valor de un premio se encuentra algo que no cumple con lo esperado.
- "TipoCasillaInvalidaException": se arroja cuando se busca el valor correspondiente a la clave "Tipo" y se encuentra un valor no conocido.

A continuación se muestran las clases que invocan cada tipo de excepción enumerada anteriormente.

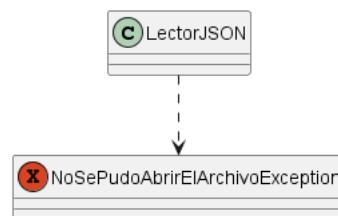


Figura 17: Excepciones que usa Lector JSON.

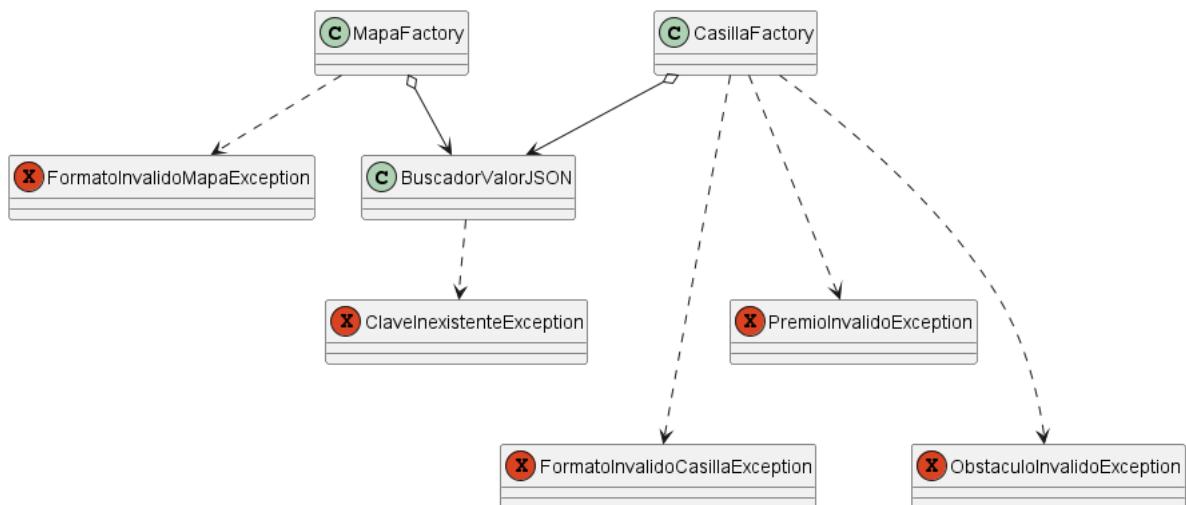


Figura 18: Excepciones que usan Mapa Factory y Casilla Factory.

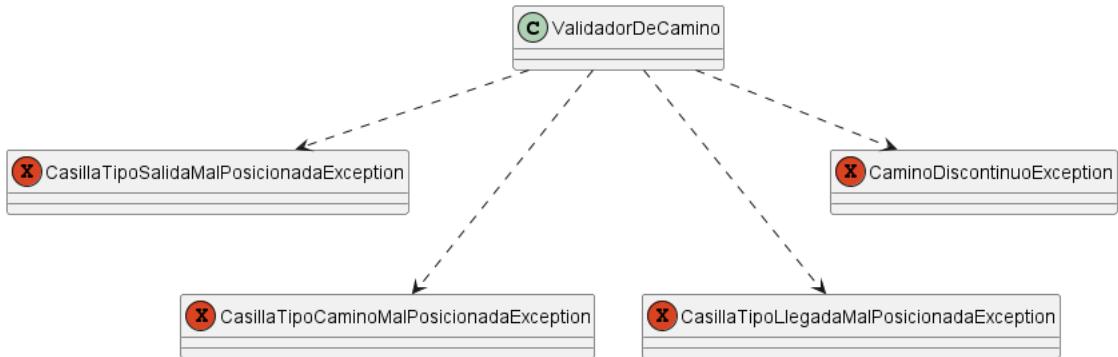


Figura 19: Excepciones que usa Validador de Caminos.

6.2. Excepciones utilizadas para el control del programa

- ”SinGanadorException”: es utilizada para controlar el flujo del programa en caso de que se haya cumplido la cantidad de turnos pre-establecidos para cada jugador y ninguno haya ganado. En ese caso se arrojará la excepción para dar fin al programa sin un ganador.

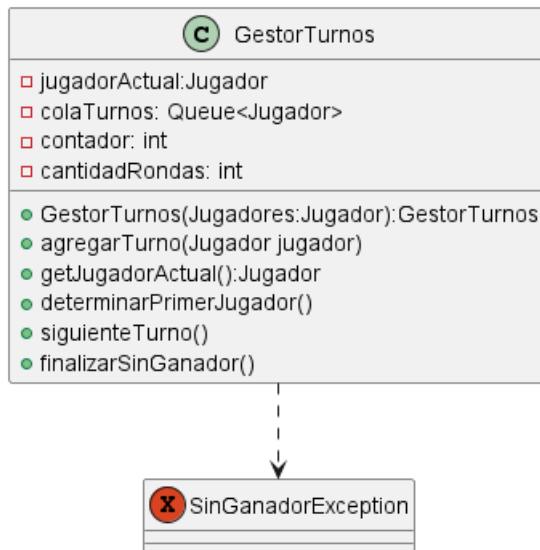


Figura 20: Excepción que usa Gestor de Turnos.

7. Diagramas de secuencia

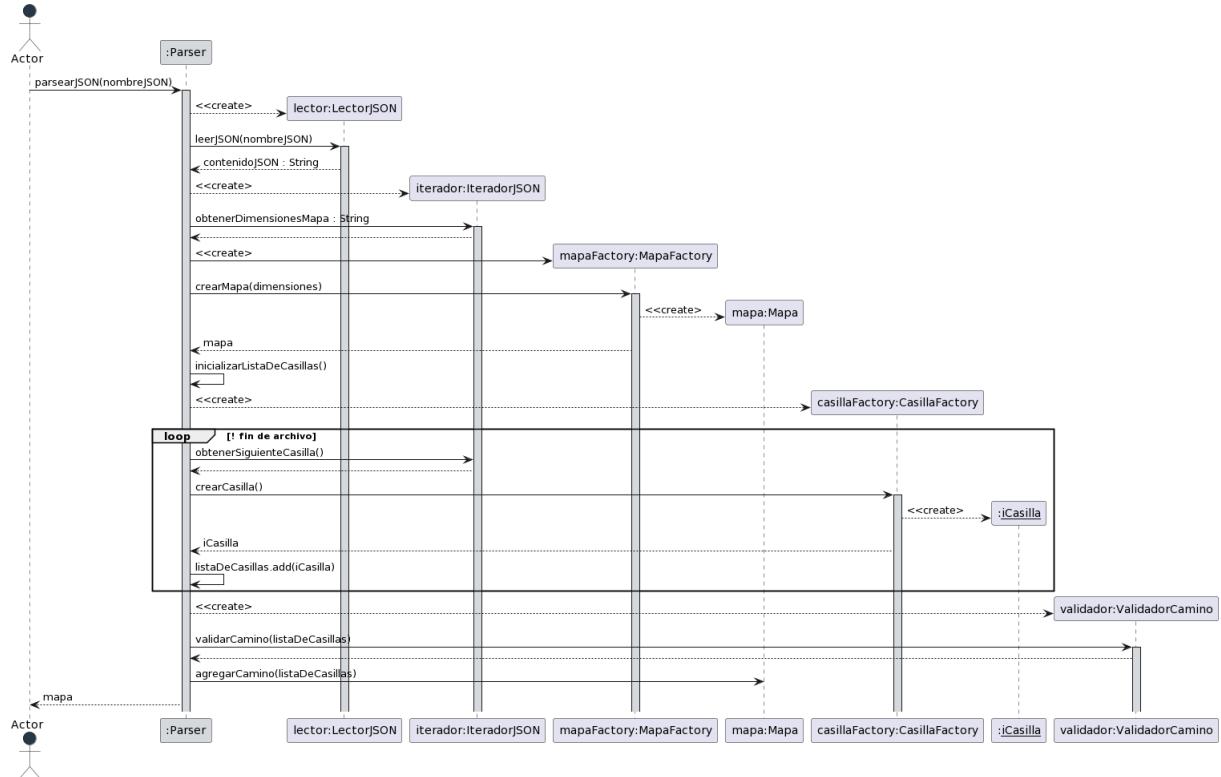


Figura 21: Diagrama de secuencia del Parser.

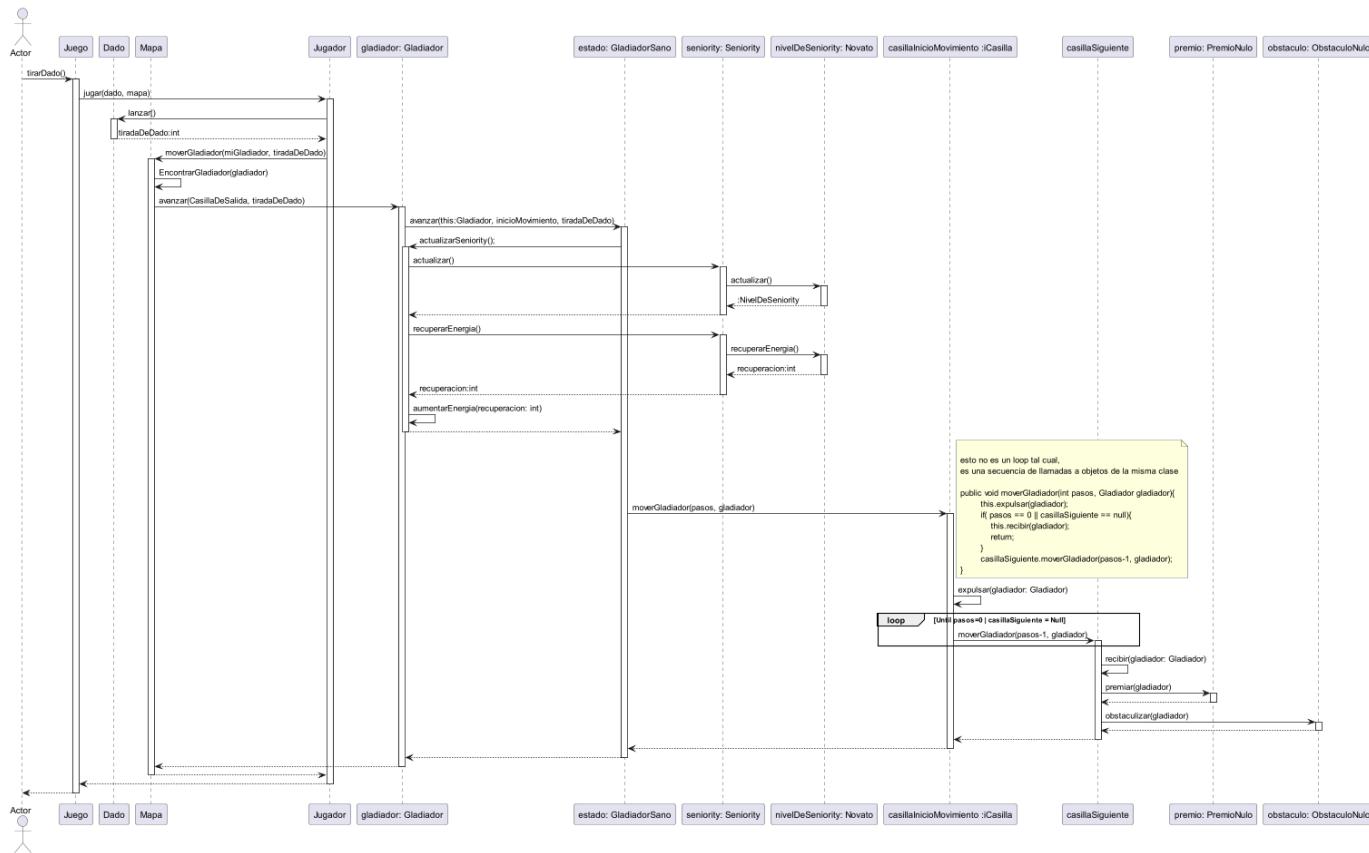


Figura 22: Diagrama de secuencia para un turno entero del Juego

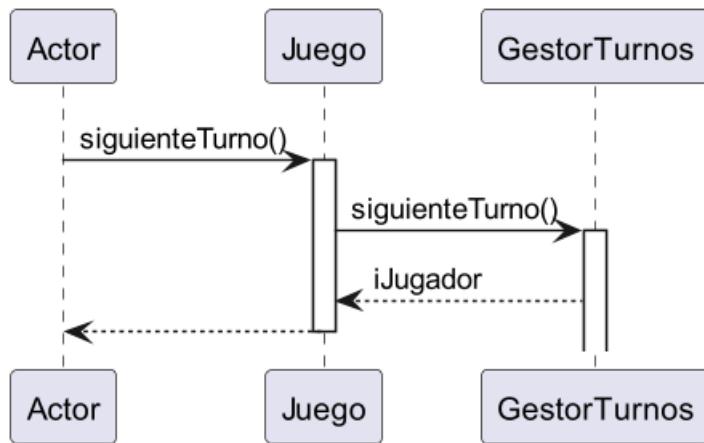


Figura 23: Diagrama de secuencia para el paso de un Turno luego de tirar el dado

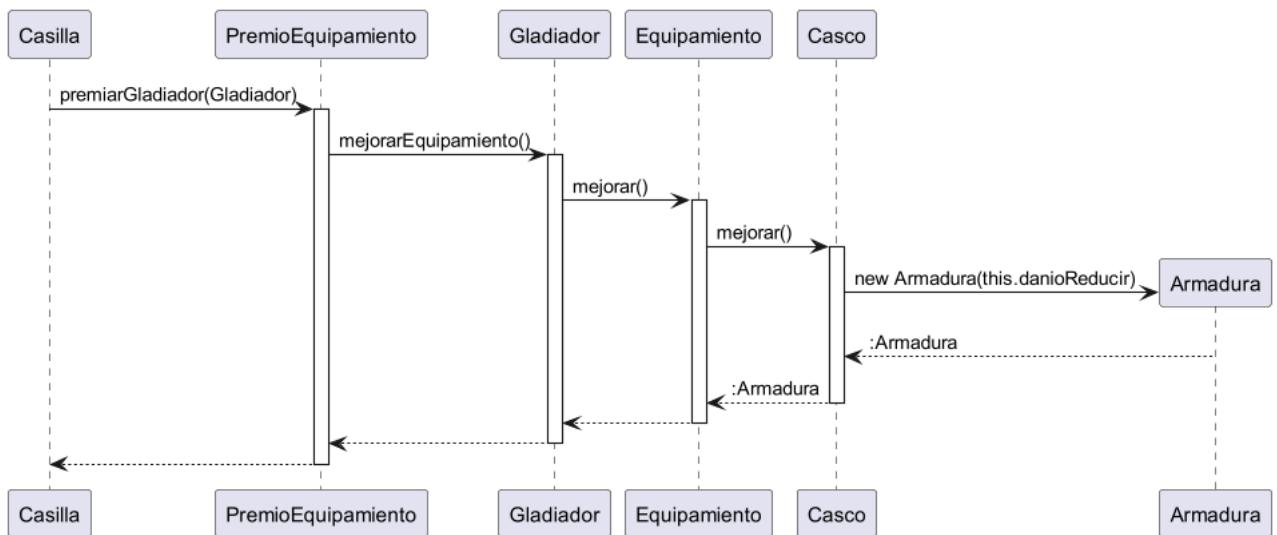


Figura 24: Gladiador cae en casilla y recibe un equipamiento

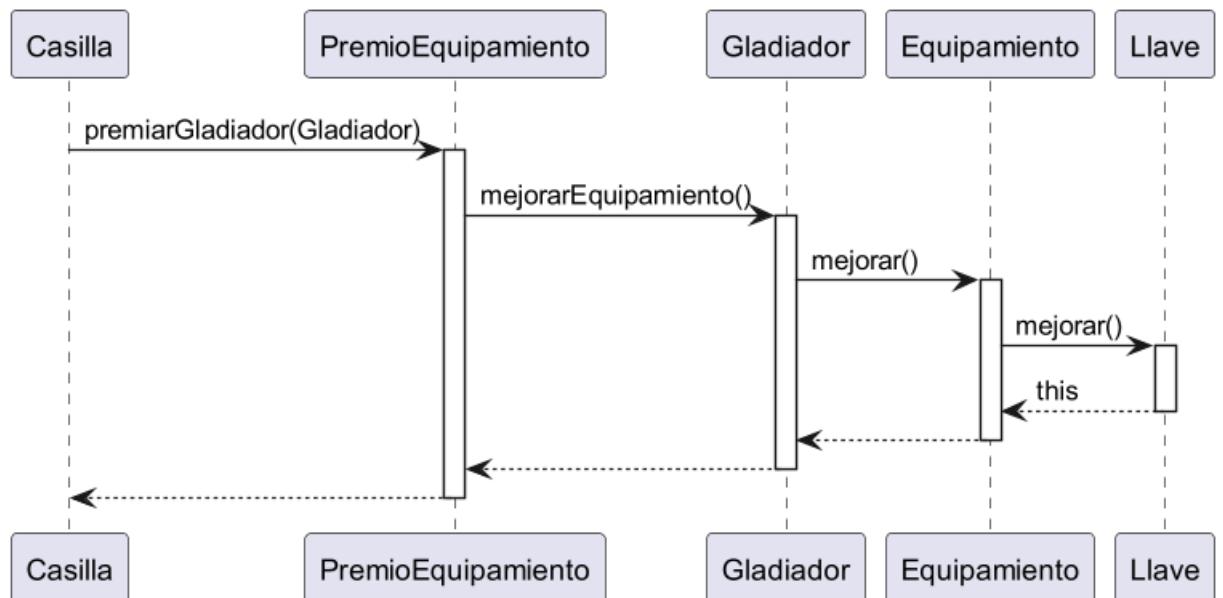
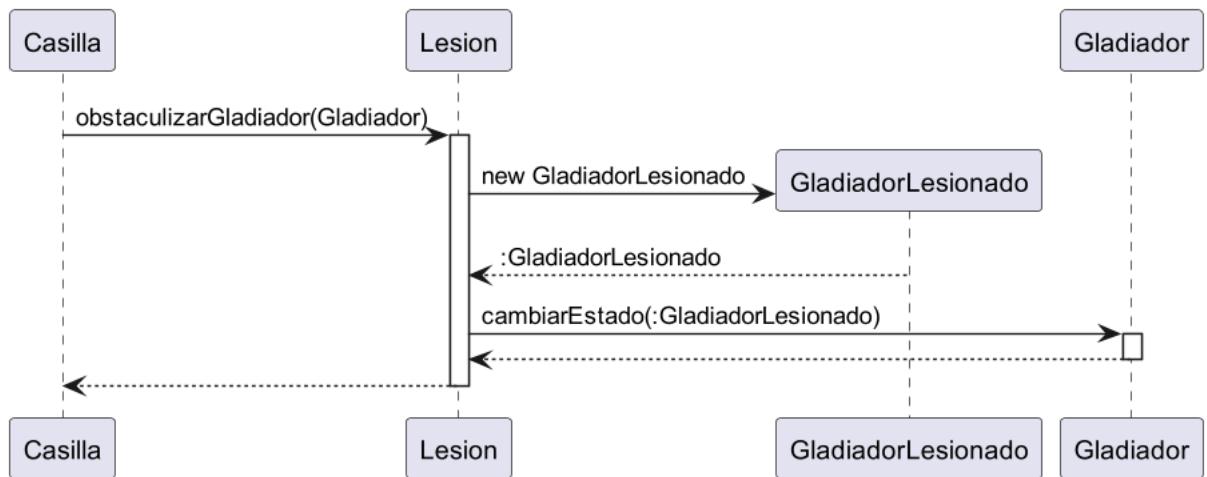
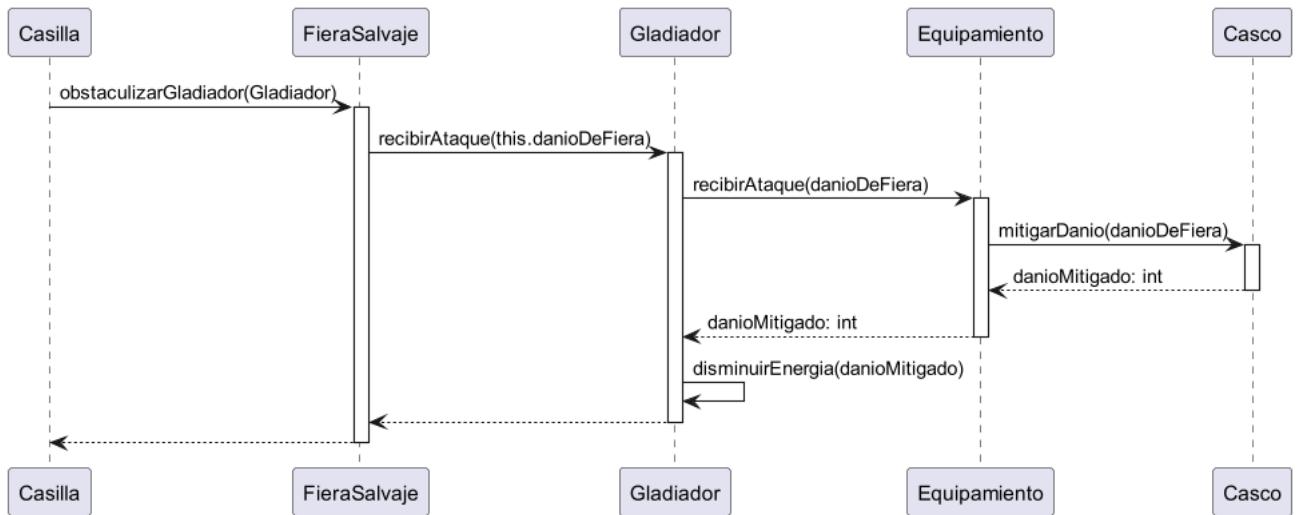


Figura 25: Gladiador cae en casilla y recibe un equipamiento pero ya tiene el ultimo Equipable existente

Figura 26: *Gladiador se lesionada*Figura 27: *Gladiador es atacado por una fiera*

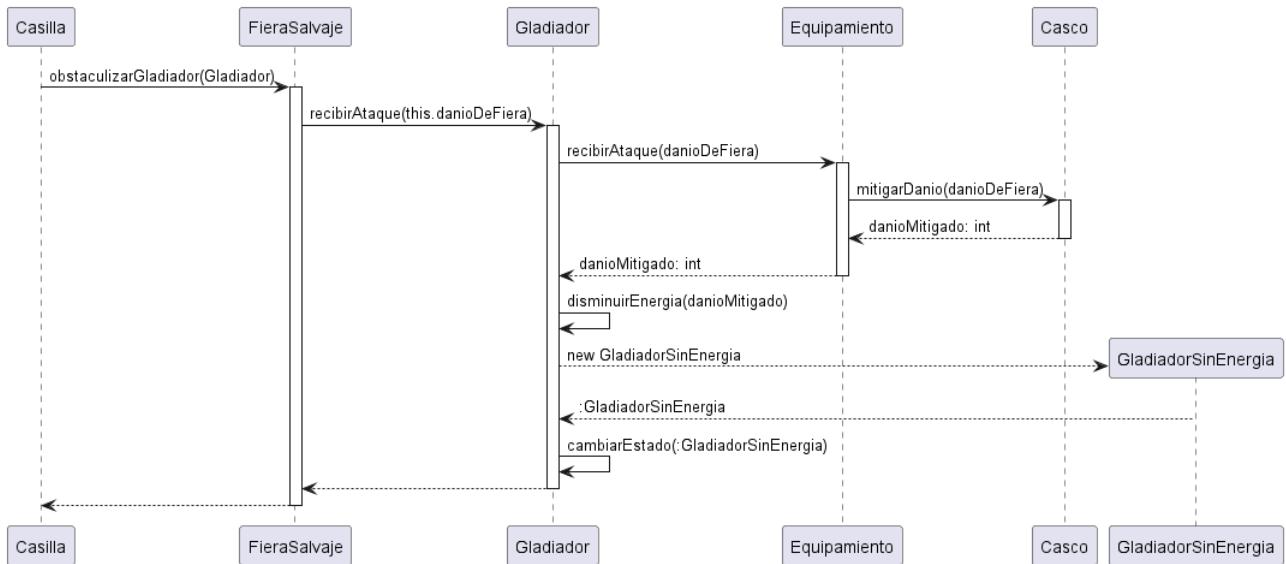


Figura 28: *Gladiador es atacado por una fiera y se queda sin energía*

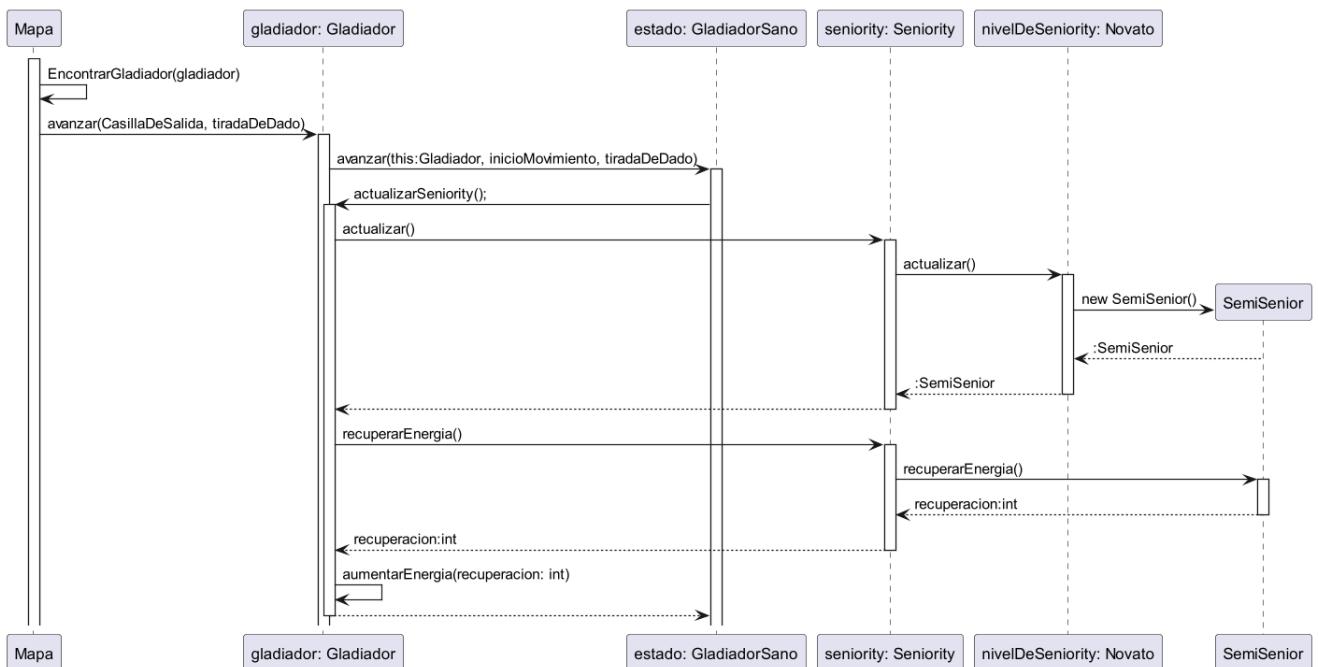


Figura 29: *Gladiador asciende de seniority y recupera energía según el seniority nuevo*

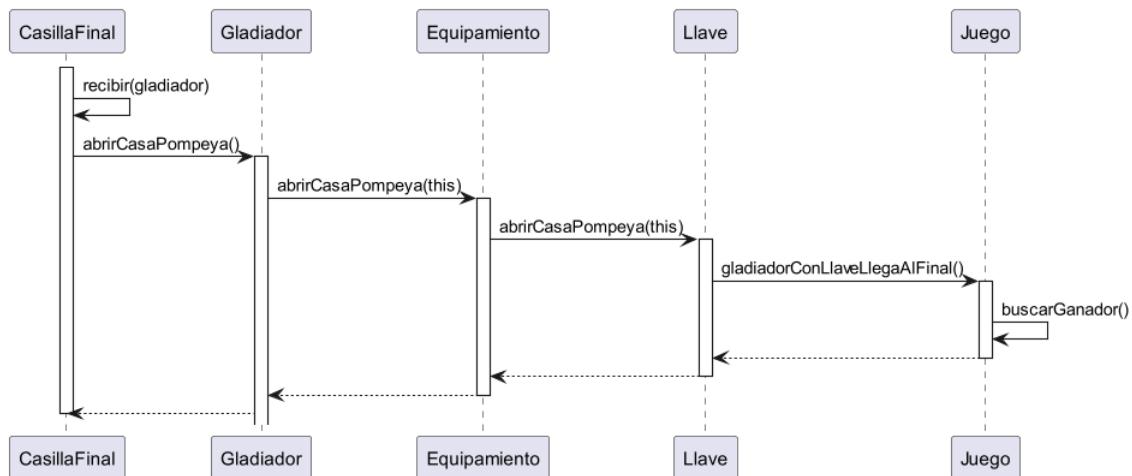


Figura 30: Gladiador llega a la casilla final con la llave

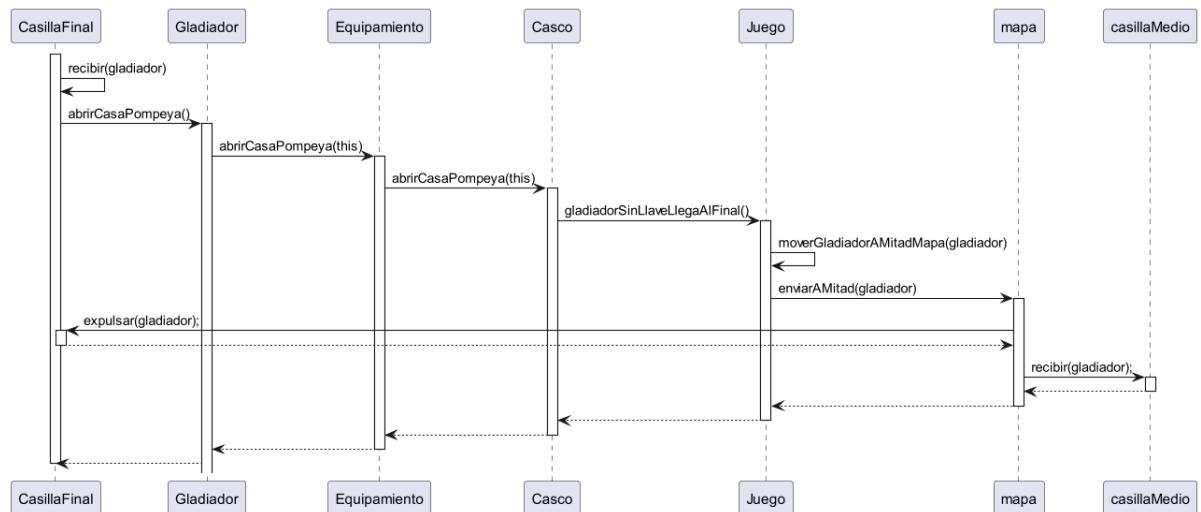


Figura 31: Gladiador llega a la casilla final sin la llave

8. Diagrama de estados

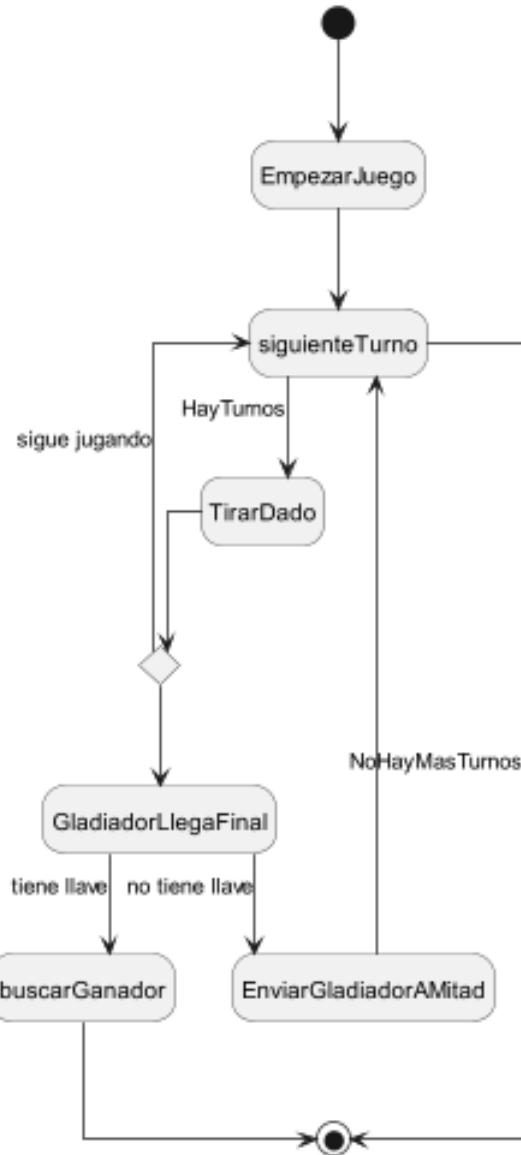


Figura 32: Estados del juego