FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

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

Alumno	**Padrón**	**Email**
Angie Isabella Valdivia Wong	103727	avaldivia@fi.uba.ar
Gabriela Alejandra Yanes Salas	108325	gyanes@fi.uba.ar
Francisco Lopez	107614	frlopez@fi.uba.ar
Ignacio Urbieta	108311	iurbieta@fi.uba.ar
Roy Fiorilo	108419	rfiorilo@fi.uba.ar

Índice

1. Introducción

Este informe detalla la solución implementada para el segundo trabajo práctico de la materia "Algoritmos y Programación III". El objetivo principal fue el desarrollo colaborativo de una aplicación, integrando de manera efectiva los conocimientos adquiridos a lo largo del curso. Para ello, se empleó Java, un lenguaje de programación de tipado estático, enfocándonos en una arquitectura de diseño orientada a objetos. Además, se adoptaron prácticas de Test Driven Development (TDD) y de Integración Continua, asegurando así la calidad y eficiencia del software desarrollado.

2. Supuestos

Dentro de la implementación de nuestro proyecto para el trabajo práctico, hemos establecido varios supuestos clave que guían el funcionamiento y las limitaciones del código. Estos son:

- **Movilidad del Jugador:** Hemos definido que el jugador solo puede desplazarse a través de las casillas de tipo "Camino". Esta decisión implica que el resto del mapa, compuesto por diversas otras casillas, está fuera del alcance del movimiento del jugador.
- **Comportamiento del Seniority:** Diseñamos el Seniority como una clase que se independiza del turno y ella misma sabe cuando cambiar.
- **Energia** Tomamos por supuesto que la energia puede ser negativa, pero siempre podra ser restablecida por el seniority a medida que pasen los turnos. Ademas podra crecer indefinidamente.
- **Premios y Obstaculos** Tomamos como supuesto que una casilla puede contener doble castigo para el gladiador (doble obstaculo) o doble recompensa (doble premio). Puede no contener ninguna de las dos, una de cada una o un solo premio/obstaculo.

3. Diagramas de clases

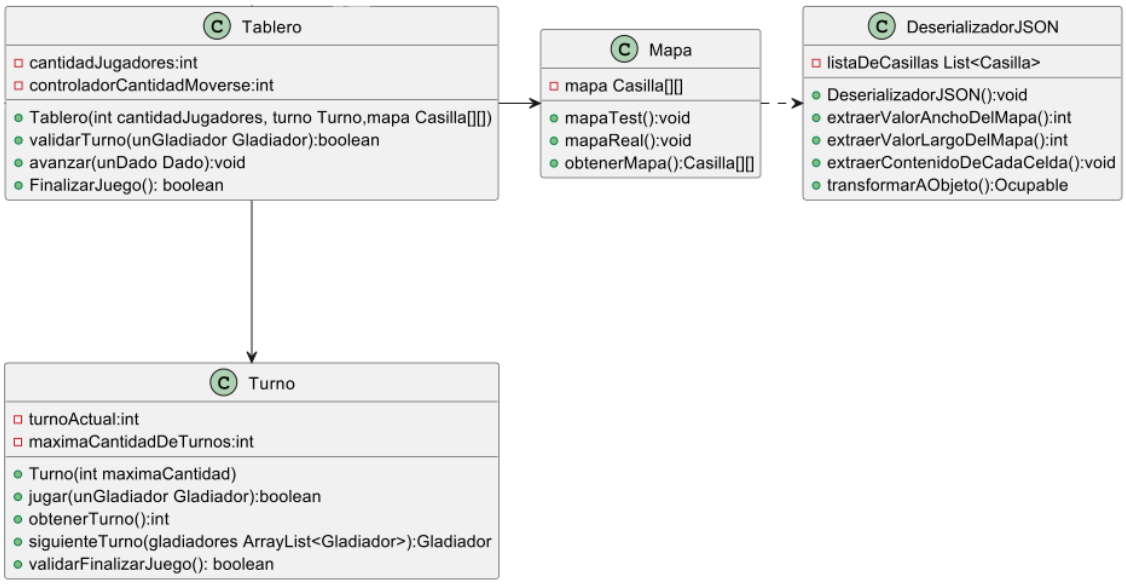

Figura 1: Relacion De Tablero con Turno, Mapa y Desrealizador JSON

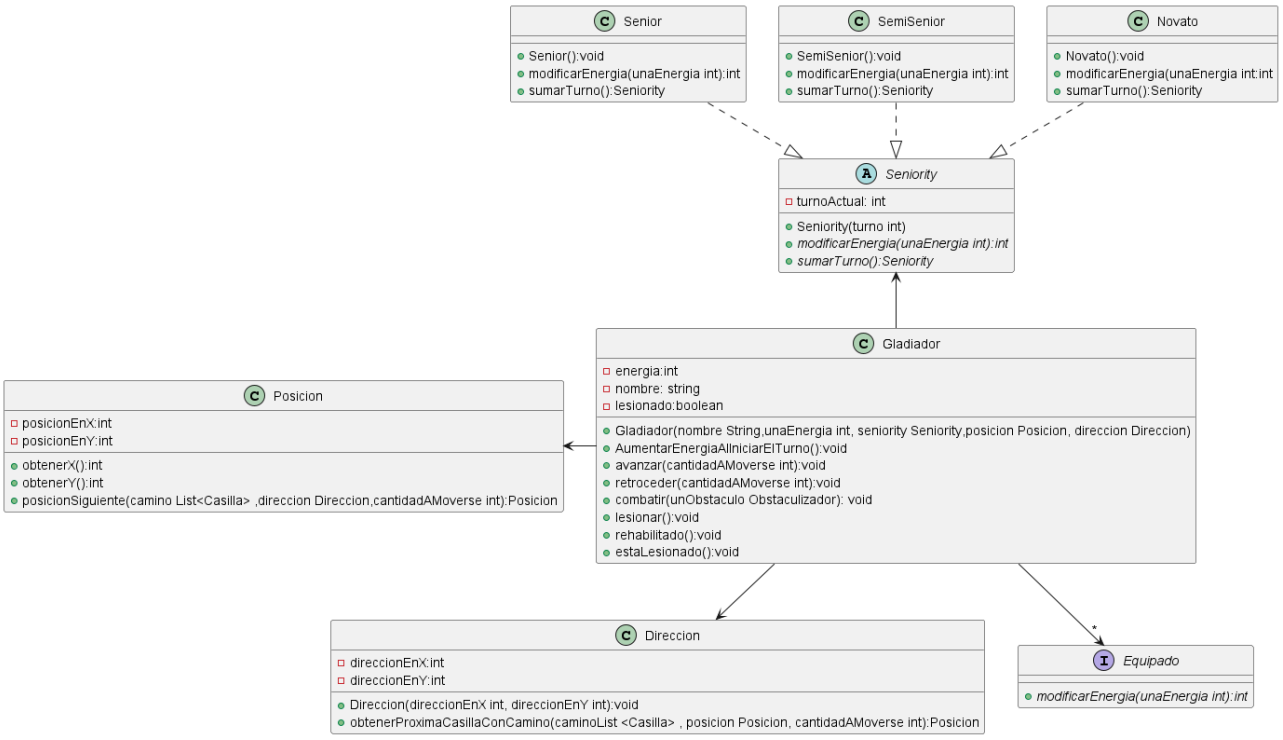

Figura 2: Relacion De Gladiador con Seniority, Direccion,Posicion,Estado y Equipado

Figura 3: Relacion De Ocupable con los equipamientos, NadaOcupacion y Equipado

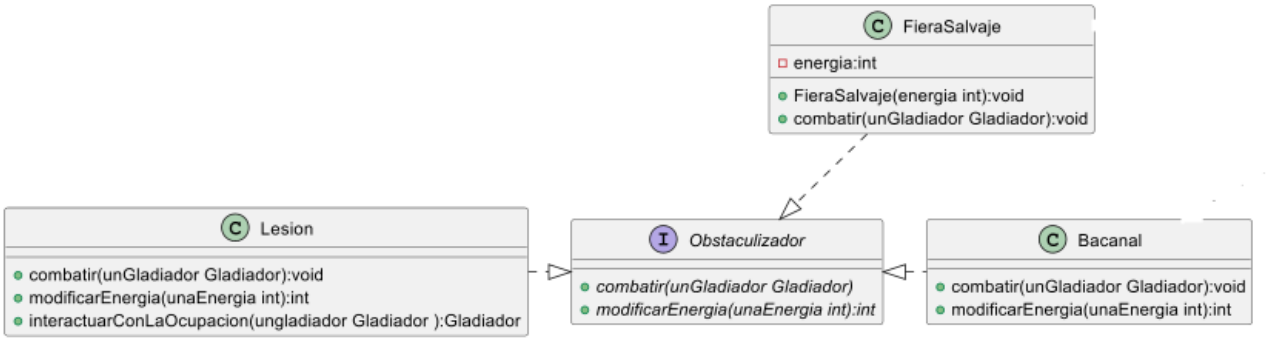

Figura 4: Relacion de los posibles obstaculos del juego

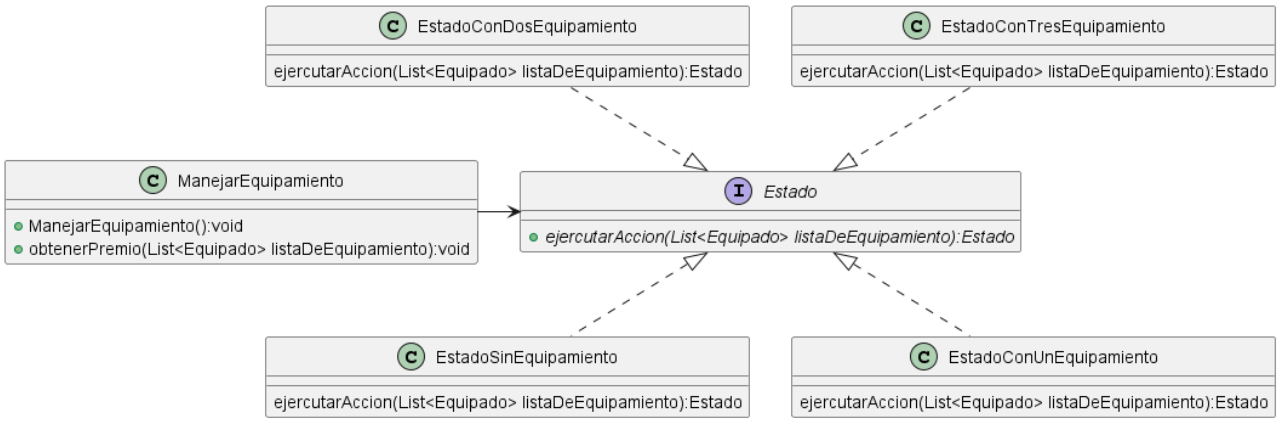

Figura 5: Aplicacion del patron state para ir cambiando el equipamiento a medida que avanza el juego

4. Diagramas de secuencia

Para ambos diagramas de secuencia, tomamos por inicializado el mapa, turno, tablero, gladiador, direccion, dado y posicion.

Como primer diagrama de secuencia, veremos la secuencia de un gladiador que cae en una casilla con una lesion y pierde su siguiente turno. Dividimos el diagrama en 2 para que sea mas legible:

UnJugadorLesionadoNoPuedeJugarElSiguienteTurno

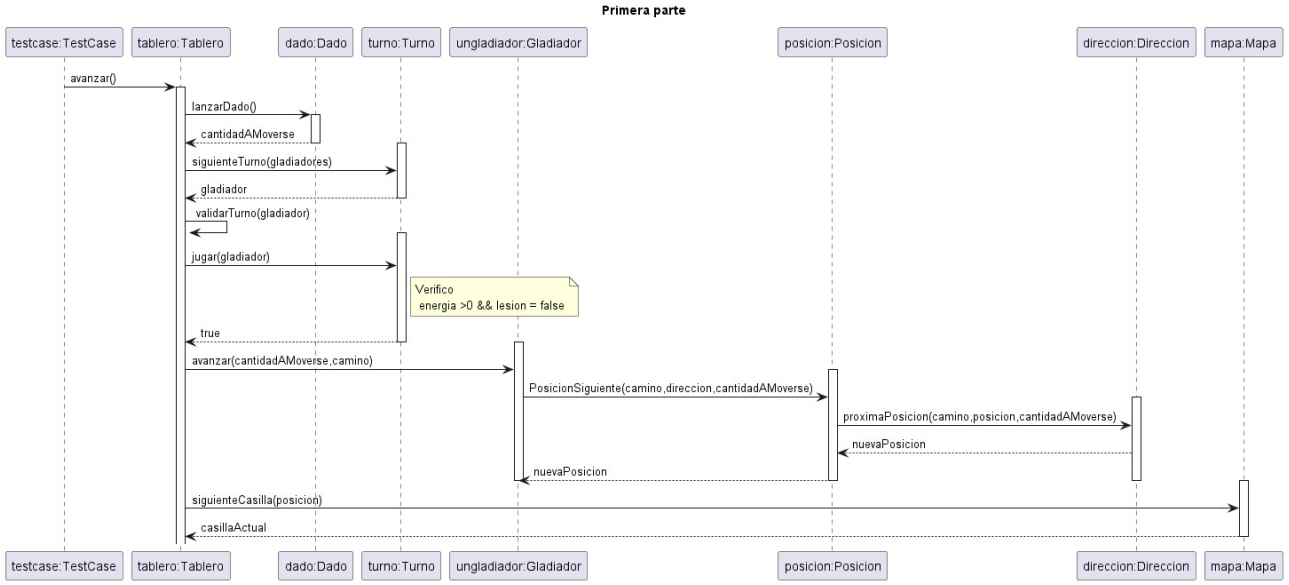

Figura 6: Parte 1

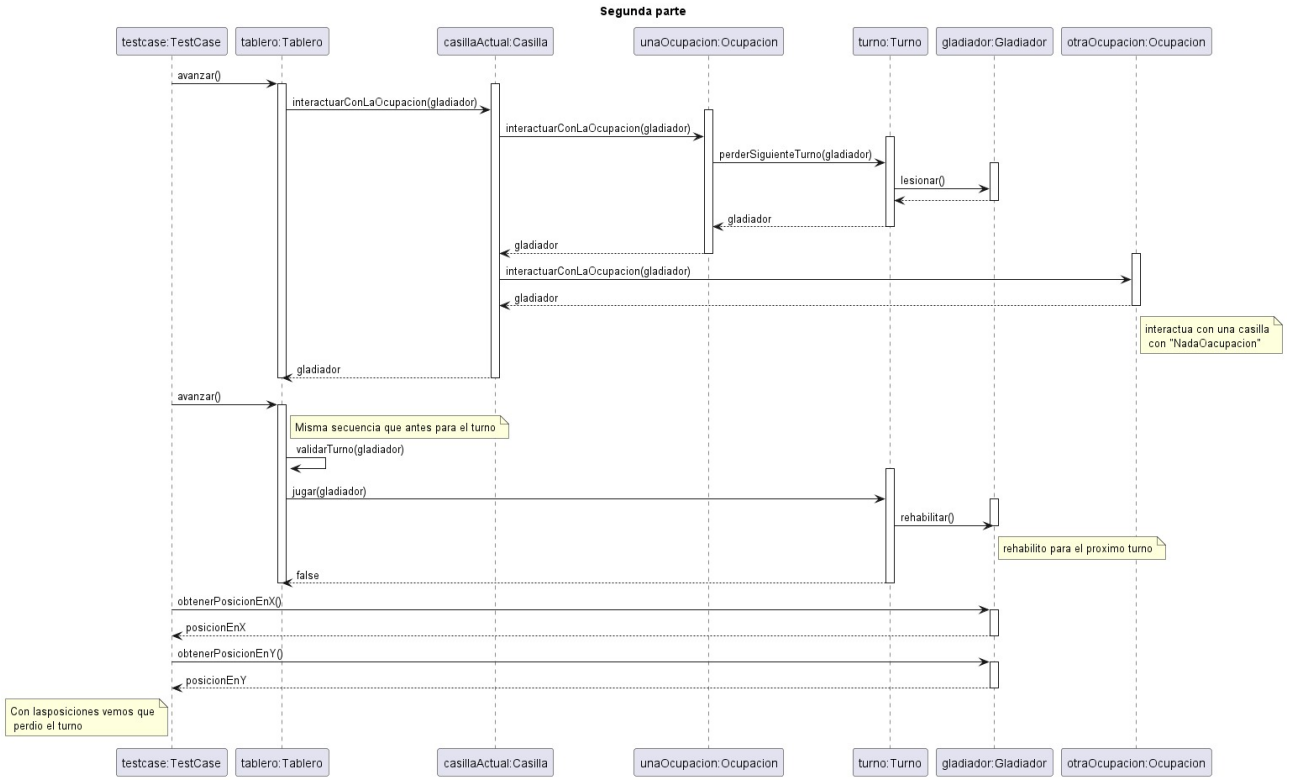

Figura 7: Parte 2

Para el segundo diagrama de secuencia, veremos un gladiador que cae en una casilla con un equipamiento y una fiera salvaje (test7 CasosDeUso1). Para este diagrama de secuencia lo dividiremos en dos partes que pueda ser mas legible.

VerificarQueSiHayUnCombateConUnaFieraSalvajeYTieneCascoPierde15Puntos

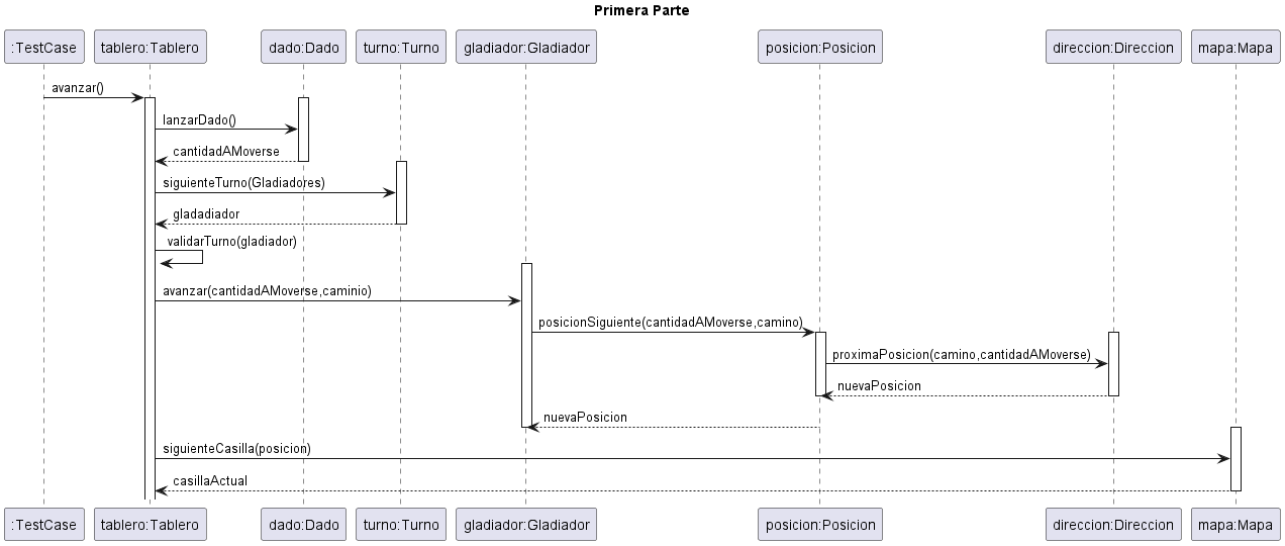

Figura 8: Parte 1

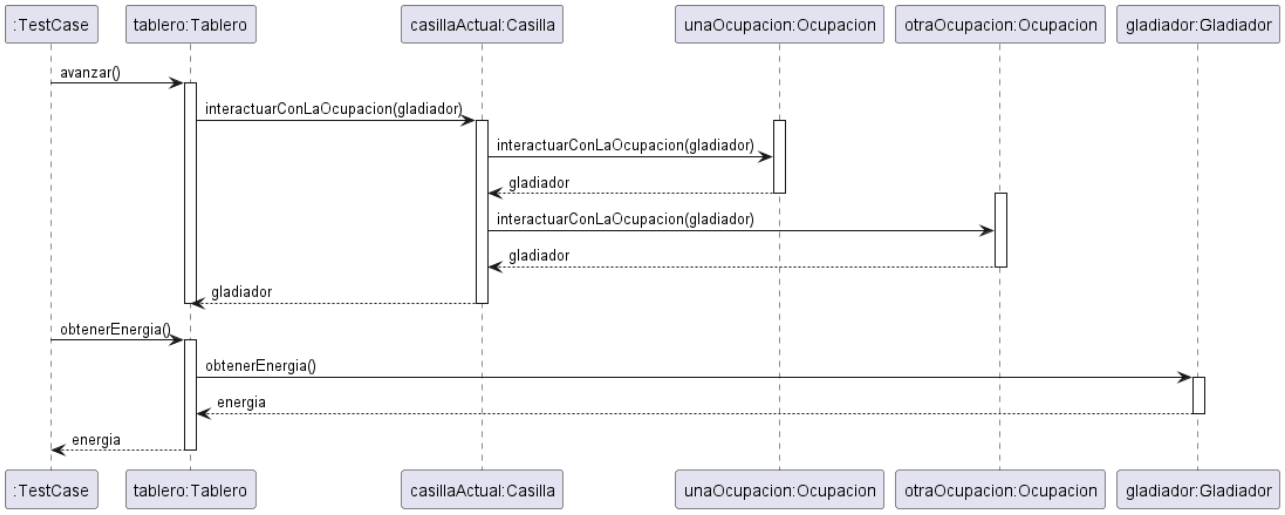

Figura 9: Parte 2

5. Diagramas de paquetes

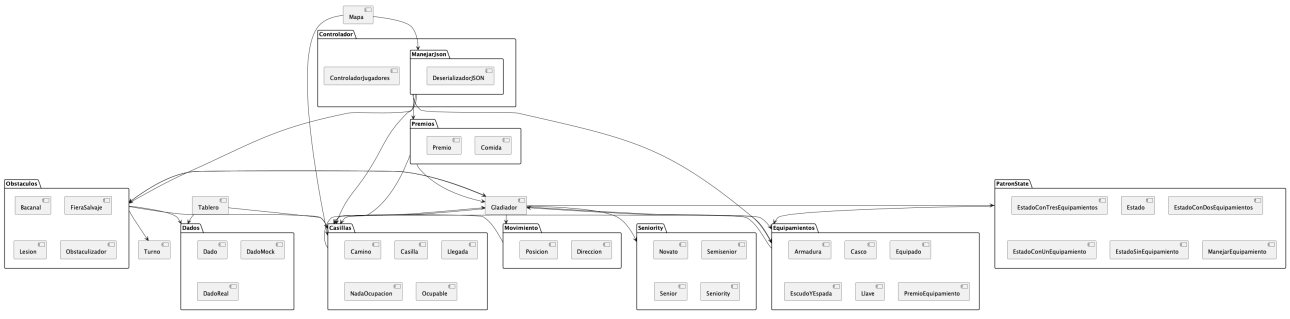

Figura 10: Diagrama de paquetes del modelo

6. Diagramas de estado

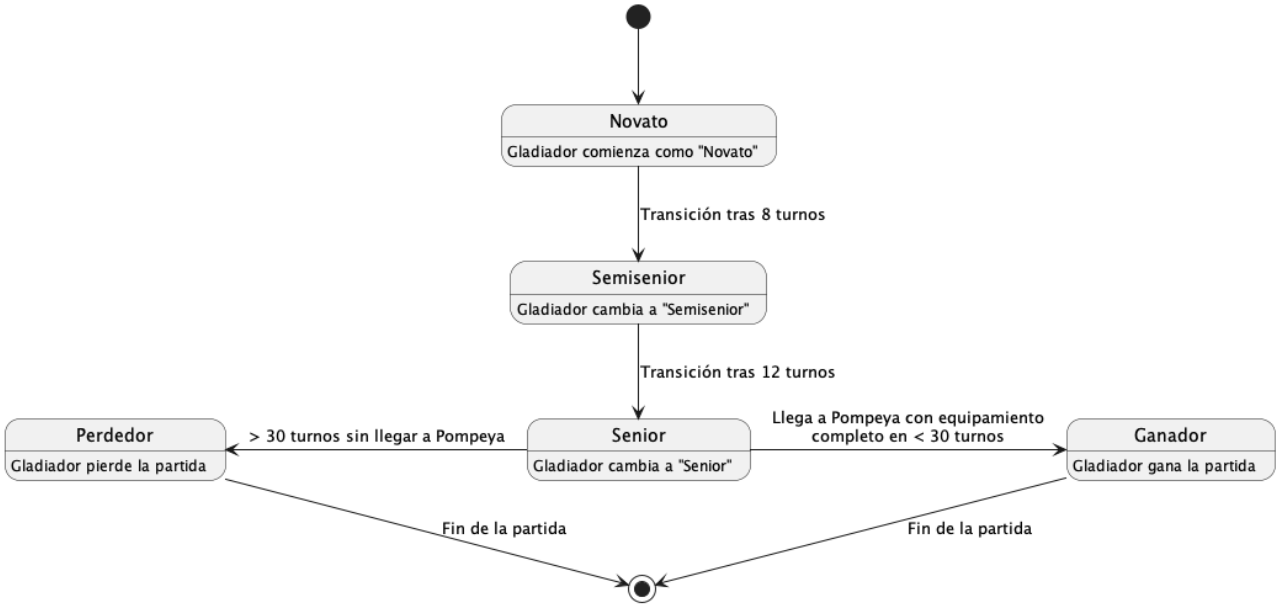

Figura 11: Diagrama de estados que muestra de forma simple los cambios de Seniority de un gladiador a lo largo de la partida.

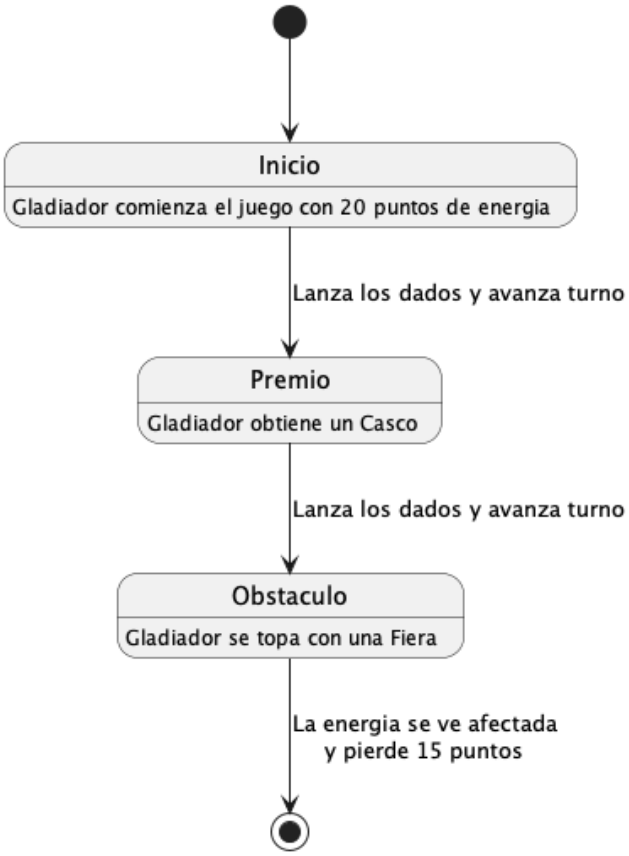

Figura 12: Diagrama de estados que muestra muestra el estado del gladiador al enfrentarse a una fiera, estando equipado con un casco como armadura, y detalla el impacto resultante en su energía.

7. Detalles de implementación

Para este trabajo practico pensamos en como diseñar varios objetos que puedan comunicarse entre si, para nuesto modelo utilizamos principalmente, un objeto Gladiador (que sera la representacion de jugador) y un tablero (el cual sera el ejecutante principal de la mayoria de las acciones de nuestro modelo).

Algunos detalles de la implementación:

7.1. Mapa

El mapa es un componente crucial de nuestro juego, concebido como una matriz que define el recorrido del gladiador. Este enfoque permite una representación clara y estructurada del espacio de juego.

- **Estructura Matricial:** Optamos por una matriz para representar el mapa, lo que facilita la navegación y la visualización del camino por el que se desplazará el gladiador.
- **Deserialización del Recorrido:** Utilizamos deserialización de un archivo .json para definir el camino en la matriz, lo que añade flexibilidad en la configuración del mapa y permite su fácil modificación sin alterar el código base.
- **Restricciones de Movimiento:** El gladiador solo puede moverse por el camino definido, lo que añade un nivel estratégico al juego y limita las opciones de movimiento a rutas específicas.

7.2. Tablero

El Tablero actúa como la entidad central de nuestro juego, integrando múltiples componentes y facilitando la interacción entre ellos.

- **Encapsulación de Componentes:** El tablero encapsula elementos clave como el mapa, la lista de jugadores, el dado y el registro del turno actual, sirviendo como el núcleo lógico del juego.
- **Gestión de Jugadores:** Permite la adición y progresión de los jugadores en el juego, manejando tanto la incorporación de nuevos participantes como su avance a lo largo del mapa.
- **Funcionalidad de Avance:** La función `avanzar` es central en la mecánica del juego. Lanza el dado, mueve a los jugadores según las casillas y valida la casilla a visitar. También determina el turno del siguiente gladiador, manteniendo el flujo del juego.

7.3. Seniority

Implementamos un sistema de "seniority" que reconoce y ejecuta el cambio de estado de los jugadores, permitiéndoles avanzar al siguiente nivel de seniority. Para construir este sistema, utilizamos herencia entre clases, basándonos en varios principios clave:

Razón de elección de Herencia

- **Reutilización de Código:** La herencia nos permite definir comportamientos y atributos comunes en una clase base, que luego son heredados por clases derivadas. Esto minimiza la redundancia y facilita el mantenimiento.
- **Jerarquía Clara:** Establecemos una jerarquía clara, donde diferentes niveles de seniority representan distintos estados de jugadores, haciendo que la relación entre ellos sea lógica y comprensible.
- **Extensibilidad:** La herencia hace que el sistema sea fácilmente extensible. La adición de nuevos niveles de seniority o la modificación de los existentes se puede lograr con cambios mínimos en el sistema general.
- **Polimorfismo:** Facilitamos el uso del polimorfismo. Las funciones que operan en la clase madre pueden manejar automáticamente cualquier clase hija, permitiendo una interacción genérica con diferentes niveles de seniority.

- **Organización del Código:** La herencia ayuda a organizar el código de manera lógica, mejorando la legibilidad y la comprensión del sistema.

A pesar de estas ventajas, somos conscientes de las limitaciones de la herencia, como el acoplamiento estrecho entre clases y la complejidad potencial.

7.4. Equipamiento

En la sección de equipamiento, hemos adoptado el patrón de diseño State para manejar los diferentes estados del equipamiento de los gladiadores, se vinculan los estados y se va actualizando el estado a medida que se van añadiendo equipamientos al gladiador.

Este patrón es particularmente adecuado para situaciones donde el comportamiento de un objeto cambia según su estado interno, lo cual es el caso de nuestro sistema de equipamiento.

Razón de elección del Patrón de Diseño State

- **Cambio de Comportamiento en Tiempo de Ejecución:** El patrón State nos permite cambiar el comportamiento del equipamiento de un gladiador durante la ejecución del juego, según el estado actual del equipamiento.
- **Reducción de Condicionales:** Al emplear este patrón, reducimos significativamente la necesidad de condicionales complejos. En lugar de usar múltiples declaraciones 'if' o 'switch', el estado actual del objeto dicta su comportamiento, lo que hace que el código sea más limpio y fácil de seguir.
- **Encapsulamiento del Estado:** Cada estado es una entidad encapsulada que contiene su propio comportamiento. Esto facilita la adición de nuevos estados o la modificación de los existentes sin afectar a otros.
- **Claridad y Mantenibilidad:** El patrón State mejora la claridad del código al separar los comportamientos asociados a diferentes estados. Esto hace que el sistema sea más fácil de mantener y extender.

Implementación:

En nuestra implementación, definimos una interfaz `Estado` que declara métodos para manejar las acciones relacionadas con el equipamiento. Cada estado concreto del equipamiento, como "EstadoConUnEquipamiento", "EstadoConDosEquipamientos" o "EstadoConTresEquipamientos", implementa esta interfaz, proporcionando su propia lógica para las acciones. El contexto del equipamiento, es decir, el objeto que representa el equipamiento del gladiador, mantiene una referencia al estado actual y delega las acciones a este estado.

Este enfoque nos permite gestionar la complejidad del sistema de equipamiento de una manera elegante y eficiente, facilitando la adición de nuevos tipos de equipamiento o estados en el futuro, sin alterar la lógica principal del juego.

7.5. Obstáculos

En nuestro juego, gestionamos tres tipos de obstáculos, cada uno con un impacto distinto en la mecánica del juego:

1. **Obstáculo Fiera Salvaje:** Reduce la energía del gladiador dependiendo de su equipamiento.
2. **Obstáculo Lesión:** Impone un estado de *lesión* al gladiador, impidiéndole jugar el próximo turno. Se rehabilita con la función `rehabilitar`.
3. **Obstáculo Bacanal:** Obliga al gladiador a consumir tragos, disminuyendo su energía en 4 puntos por cada trago.

Para los obstáculos, implementamos una interfaz `Obstaculizador` con métodos como `combatir` y `modificarEnergia`. Creamos clases para cada tipo de obstáculo (Bacanal, FieraSalvaje, Lesion) siguiendo esta interfaz.

Razón de elección de Interfaces sobre Herencia

- **Flexibilidad en Implementación:** Utilizando interfaces, proporcionamos una estructura flexible que permite a cada clase de obstáculo implementar sus propios métodos específicos, manteniendo un contrato común.
- **Fomenta el Bajo Acoplamiento:** Esta estrategia reduce el acoplamiento entre las diferentes clases de obstáculos y otras partes del sistema, facilitando modificaciones y extensiones futuras.
- **Mejora de la Mantenibilidad:** Al separar la definición del comportamiento (interfaz) de su implementación (clases concretas), hacemos que el código sea más fácil de entender y mantener.
- **Principio de Inversión de Dependencia:** Al depender de abstracciones (interfaces) en lugar de clases concretas, nuestro diseño se alinea mejor con los principios SOLID, especialmente el de Inversión de Dependencia.

Estas decisiones de diseño reflejan nuestro enfoque en crear un sistema robusto y escalable, donde los cambios en un tipo de obstáculo no afecten a los demás, y se puedan añadir nuevos obstáculos con facilidad.