FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

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

Nombre	Padrón	Email
Barreto, Sair	109492	gbarreto@fi.uba.ar
Galindez, María	105741	mgalindez@fi.uba.ar
Germinario, Agustina	109095	agerminario@fi.uba.ar
Pratto, Florencia	110416	fnpratto@fi.ub.ar
Kraglievich, Sebastián	109038	skraglievich@fi.uba.ar

Índice

1. Introducción

El presente informe reune la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III. El mismo consiste en desarrollar una aplicación relacionada a un juego de tablero por turnos entre 2-6 jugadores ambientado en el imperio romano. Nuestro objetivo es aplicar los conceptos explicados a lo largo de la materia, utilizando un lenguaje de tipado estático llamado Java, con un diseño del modelo orientado a objetos. El producto final siendo una aplicación completa, incluyendo el modelo de clases, sonidos e interfaz gráfica.

2. Supuestos

Si bien para algunos aspectos la especificacion era clara, para otros se tuvieron que adoptar los siguientes supuestos:

- Cuando un gladiador cae en una celda, primero es afectado por el premio y luego por el obstaculo.
- Un gladiador no puede tener energía negativa, el límite de la energía es cero.
- No hay límite de energía positiva del gladiador.
- Los nombres de todos los jugadores deben ser distintos.
- En el caso de que el gladiador llegue al final sin estar completamente equipado, al volver al medio no es afectado por el premio y obstaculo de la celda media.
- El gladiador mejora su seniority y recibe tal energía cuando finaliza de moverse.

3. Diagramas

3.1. Diagramas de clase

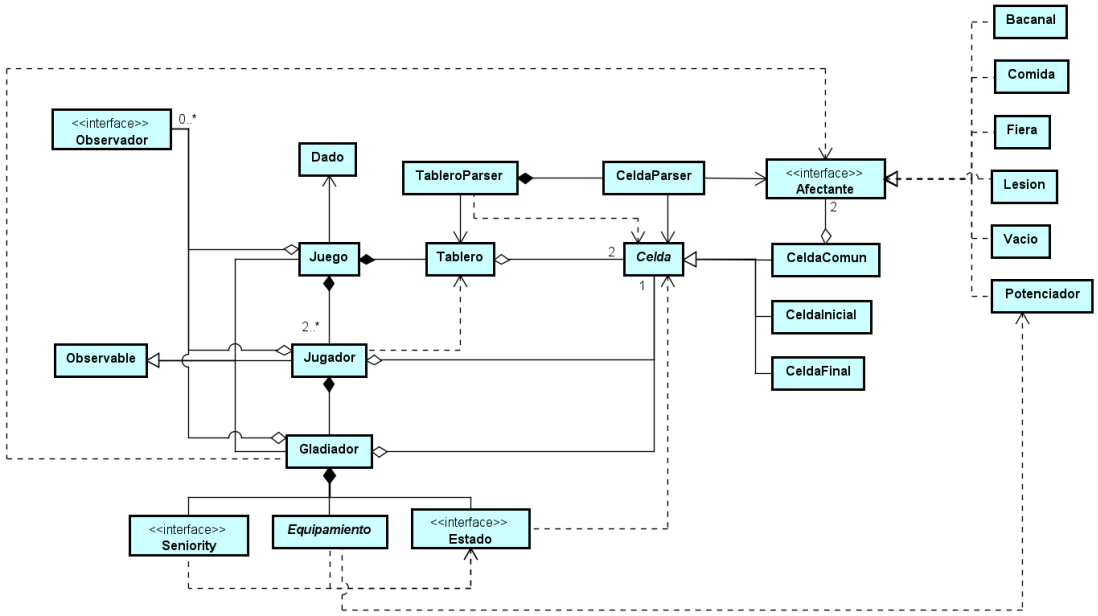

Figura 1: Diagrama de clases general del modelo.

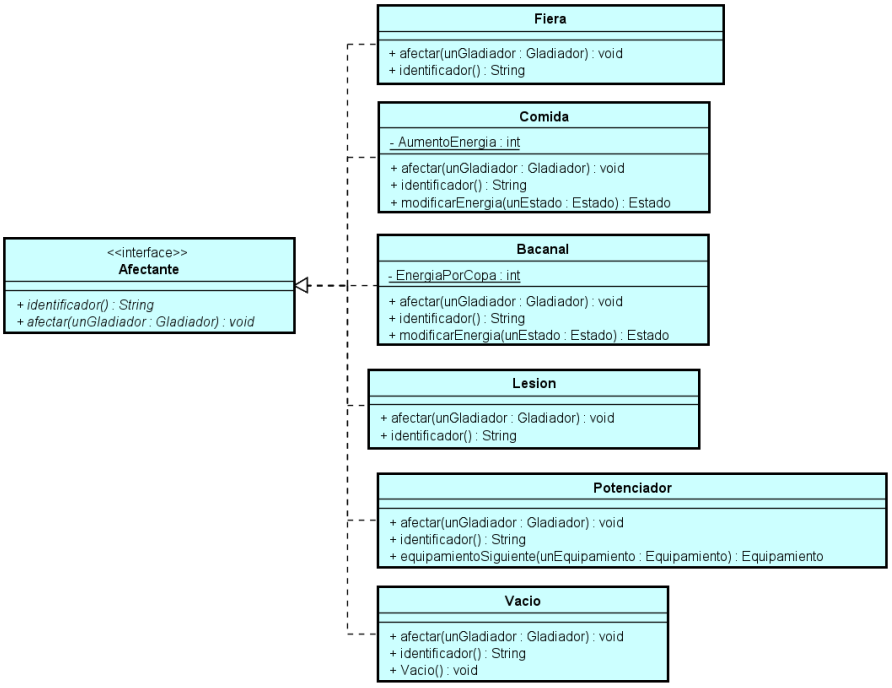

Figura 2: Diagrama de la interfaz Afectante.

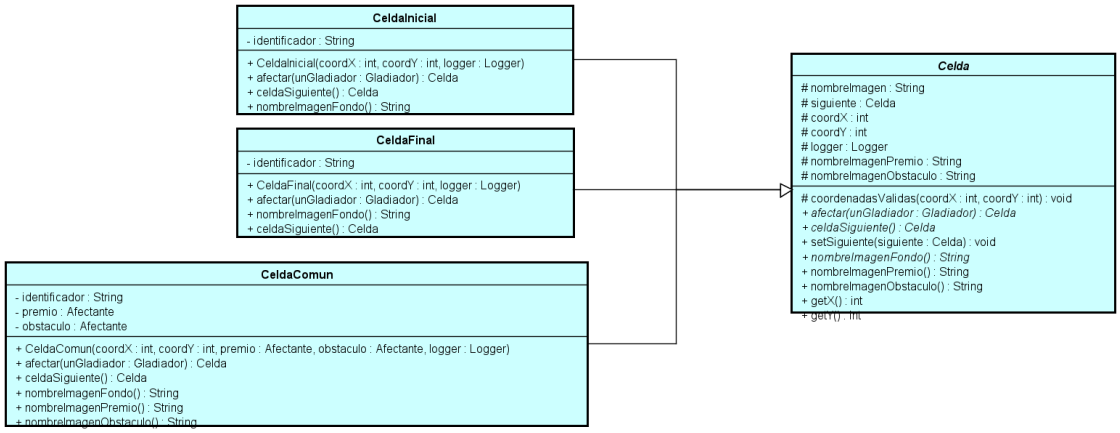

Figura 3: Diagrama de la relación de herencia de Celda.

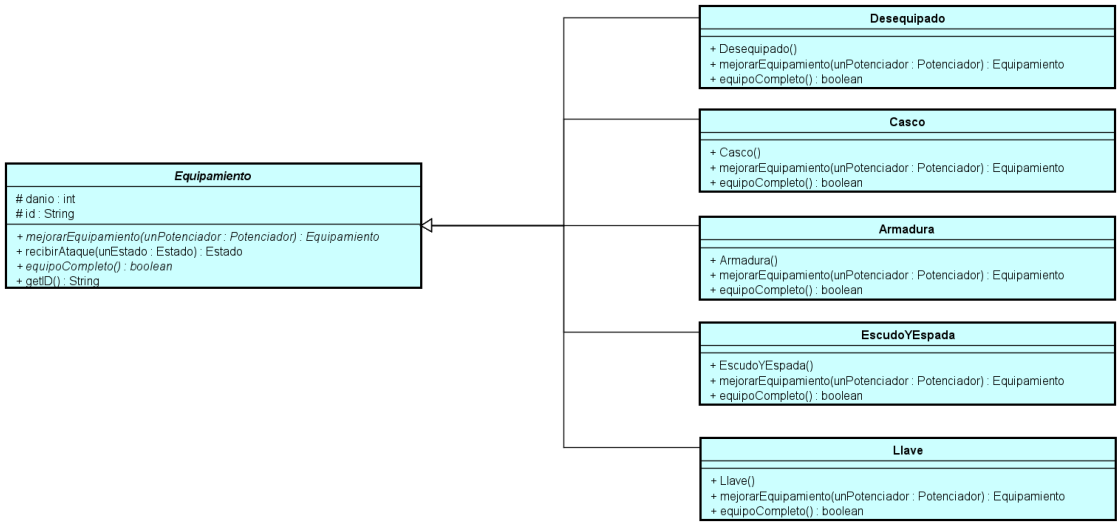

Figura 4: Diagrama de la relación de herencia de Equipamiento.

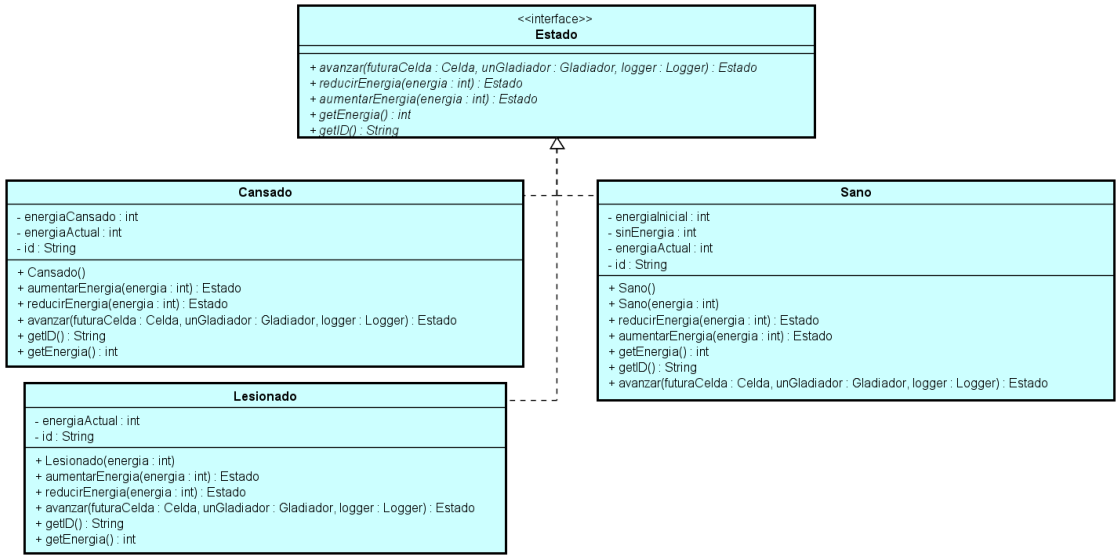

Figura 5: Diagrama de la interfaz Estado.

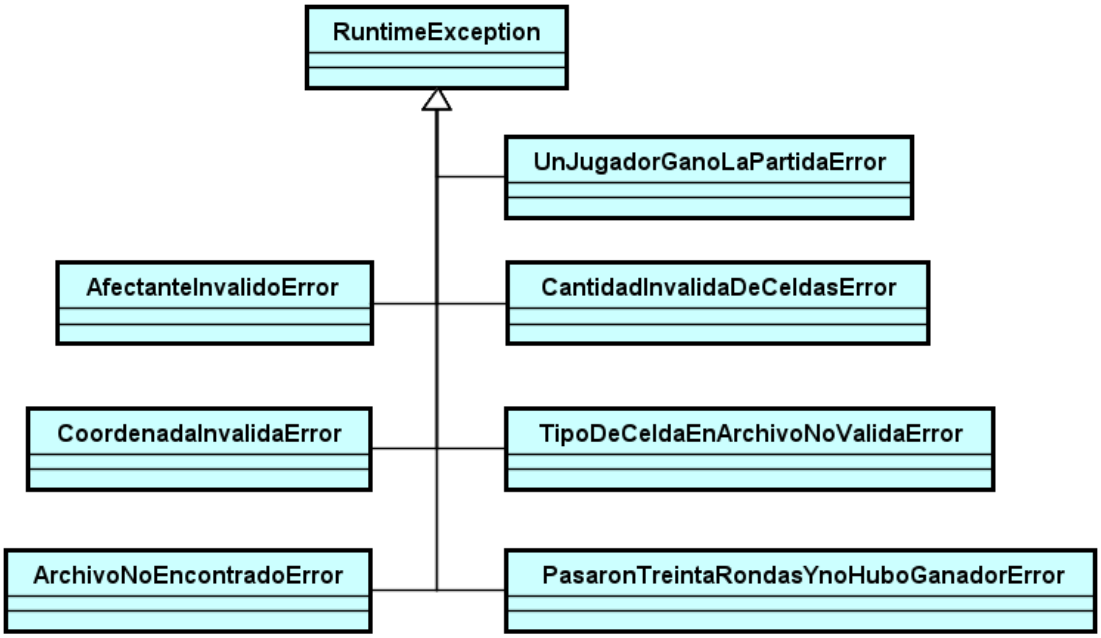

Figura 6: Diagrama de las Excepciones.

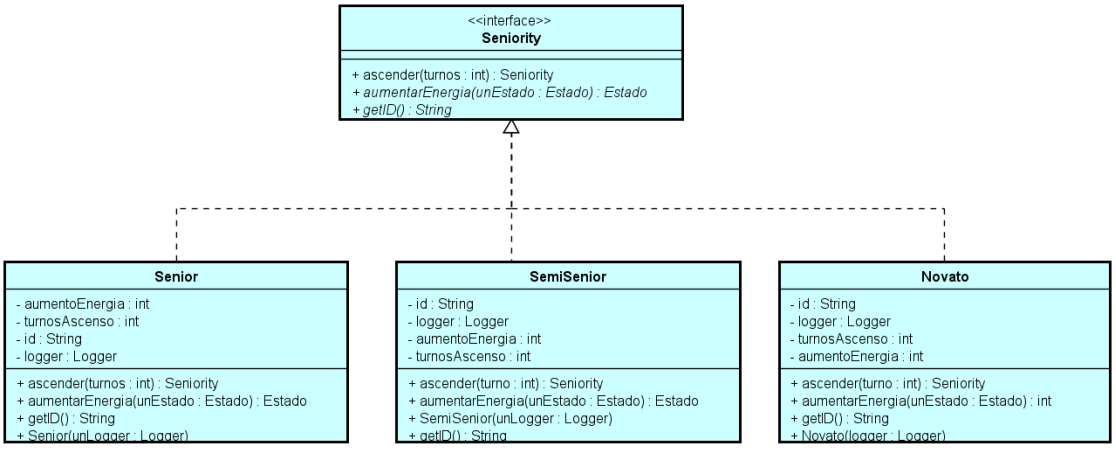

Figura 7: Diagrama de la interfaz Seniority.

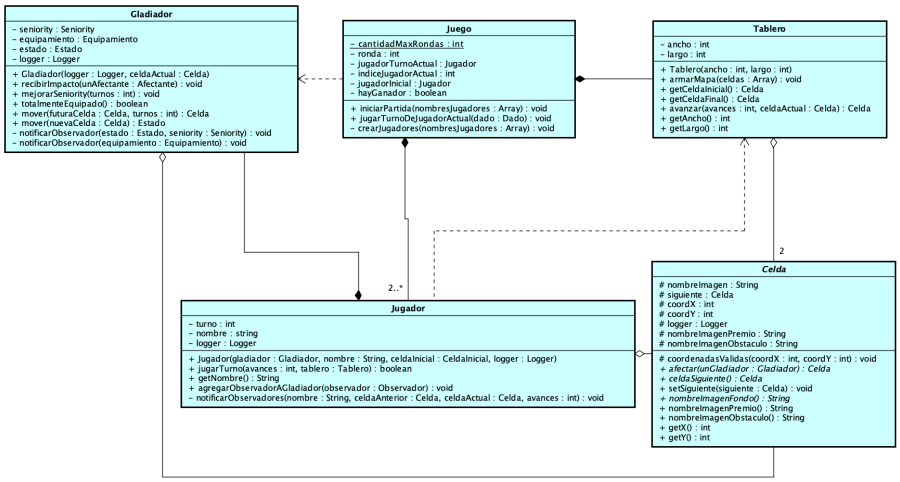

Figura 8: Diagrama de la relación entre Juego, Tablero, Celda, Jugador y Gladiador

3.2. Diagramas de secuencia

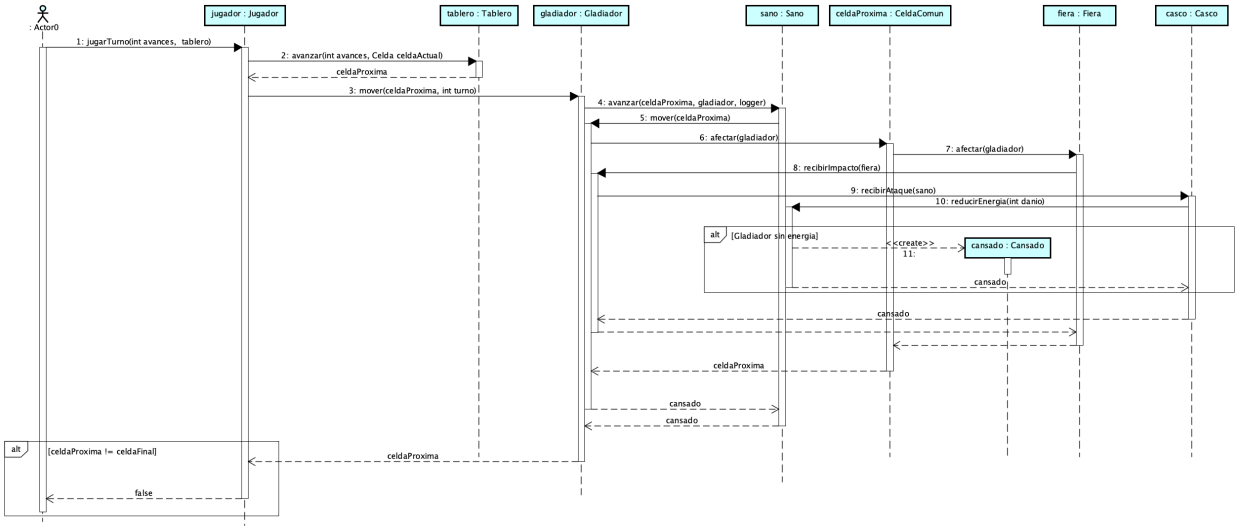

Figura 9: El gladiador con Casco cae en una celda con una Fiera.

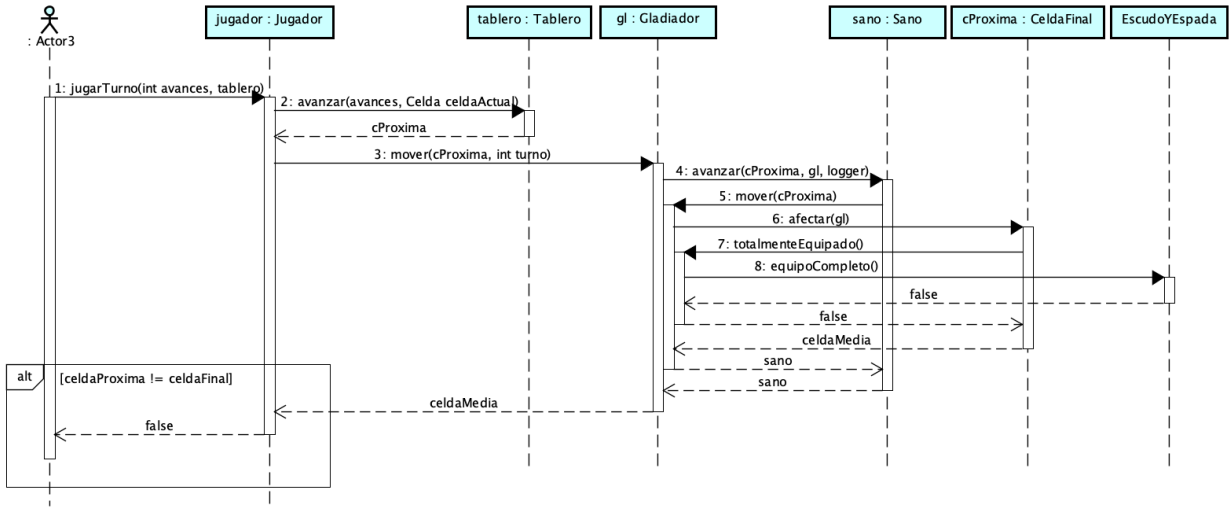

Figura 10: Un gladiador llega al final sin estar completamente equipado.

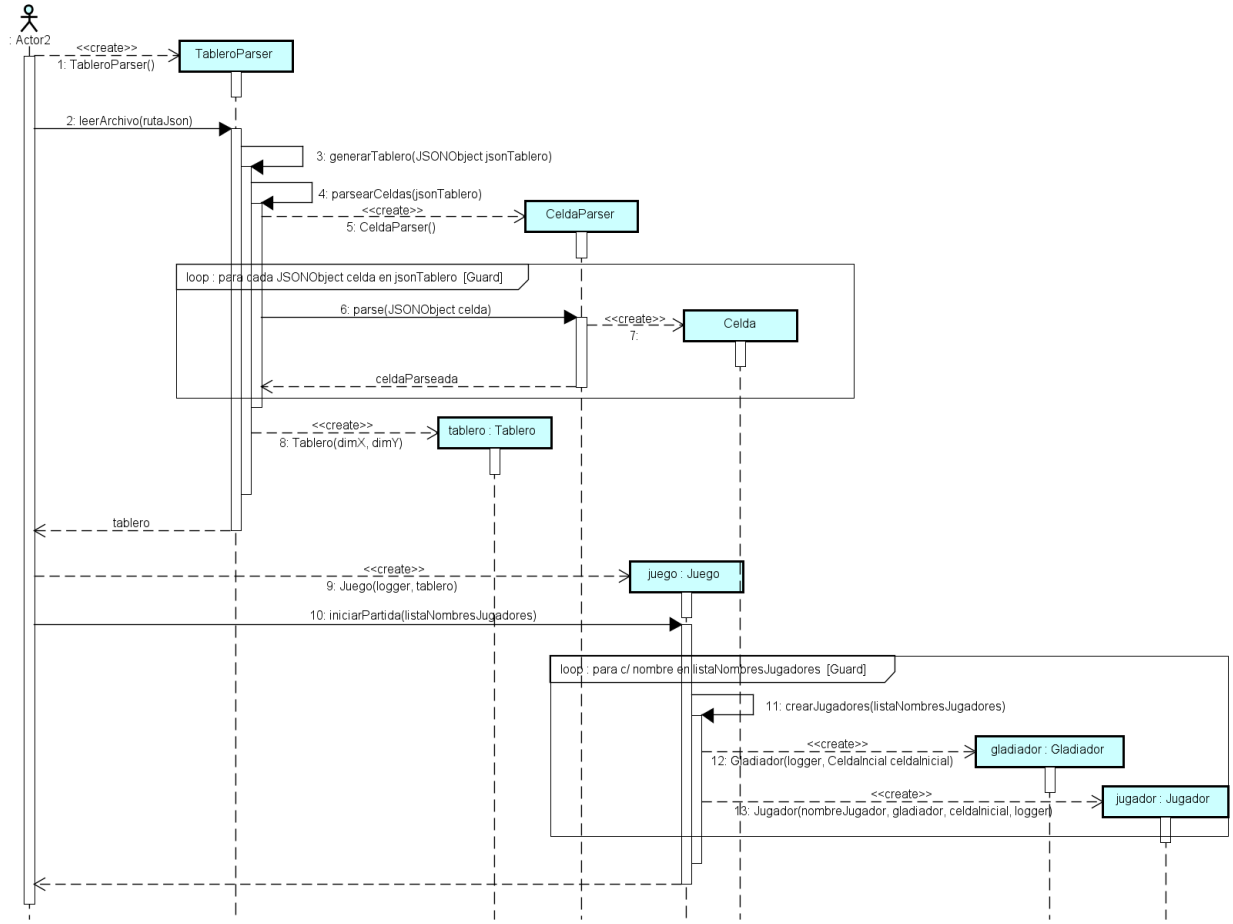

Figura 11: Inicialización de las clases Tablero y Juego.

Figura 12: Pasan 30 turnos y no hay ganador.

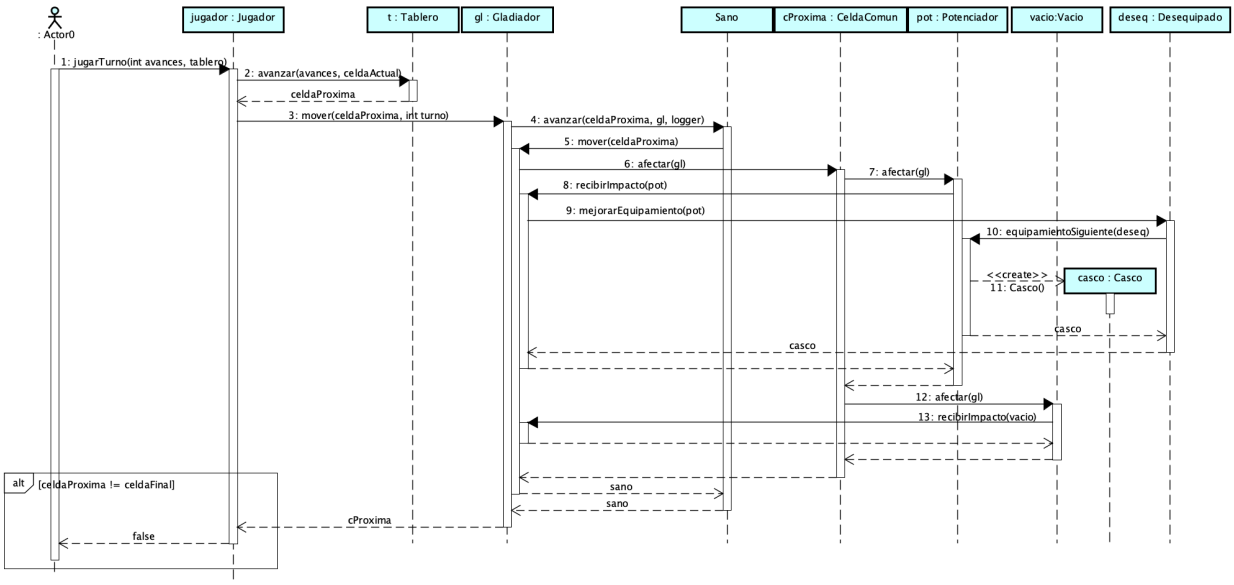

Figura 13: Un Gladiador desequipado cae en una Celda con Potenciador.

3.3. Diagrama de paquetes

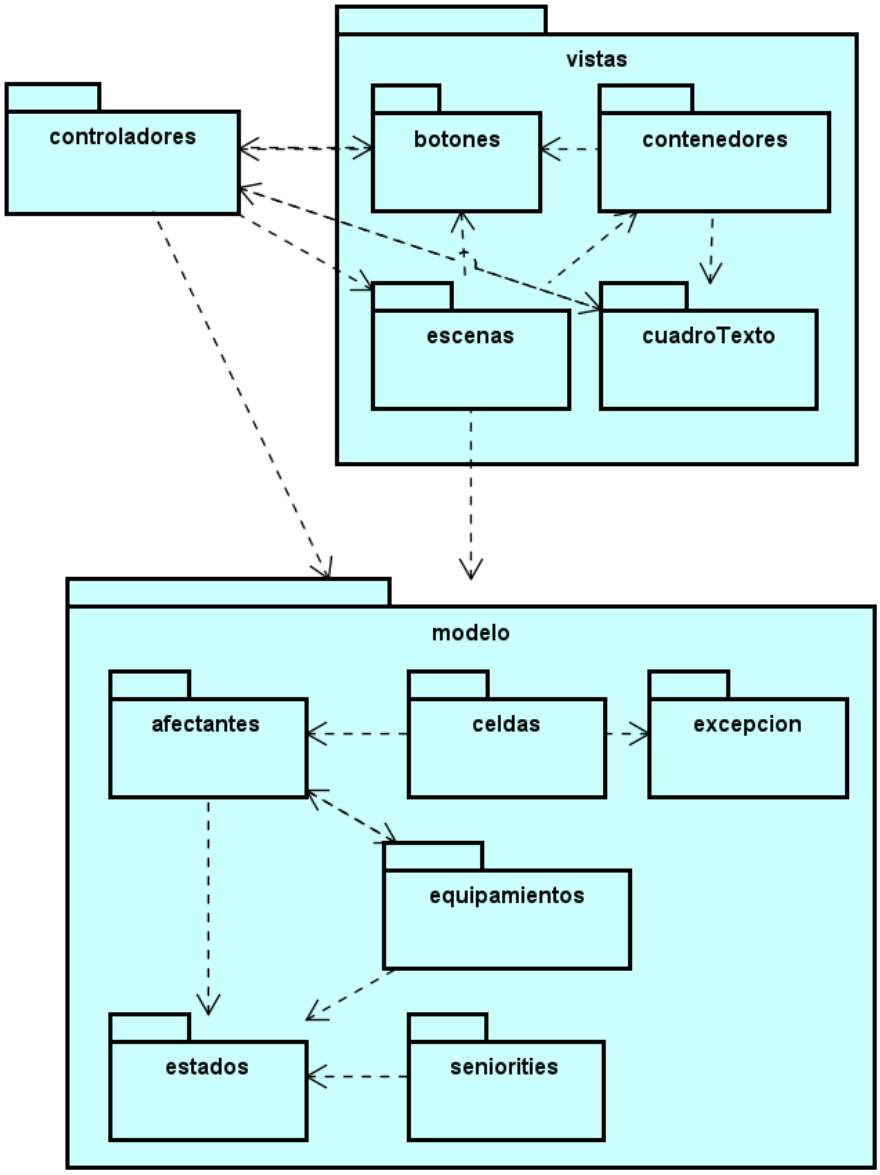

Figura 14: Diagrama de Paquetes.

3.4. Diagramas de estado

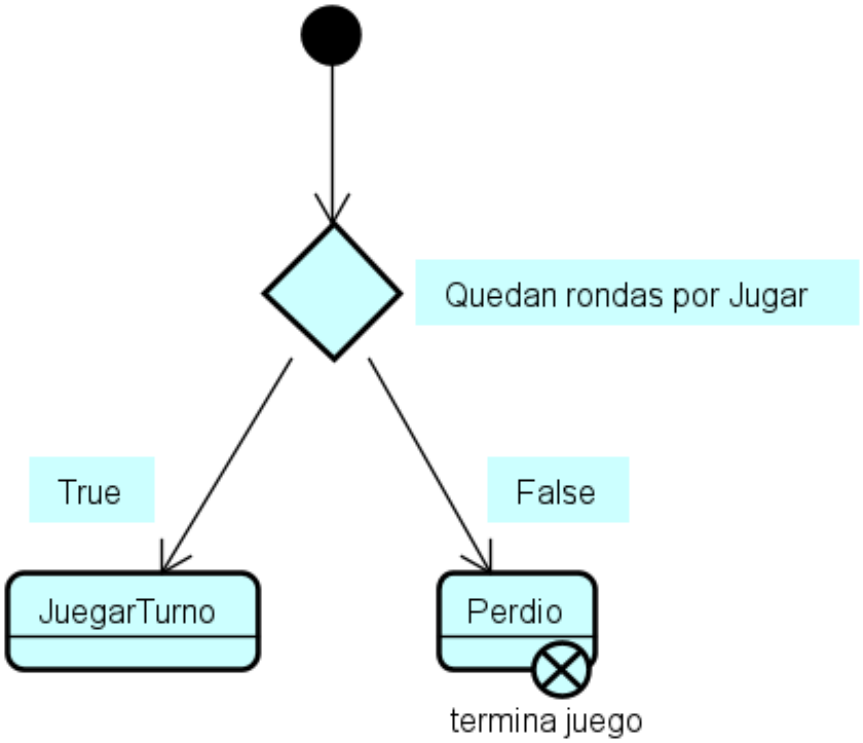

Figura 15: Fin de la partida

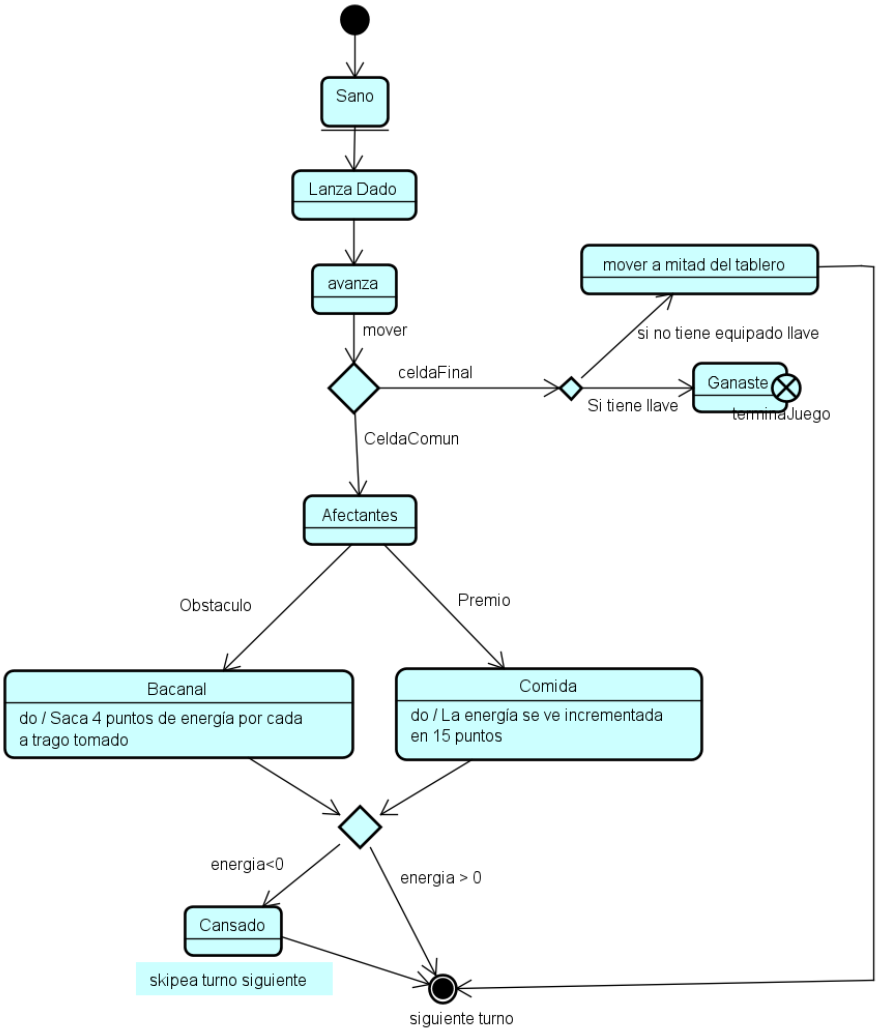

Figura 16: Jugar turno caso Bacanal

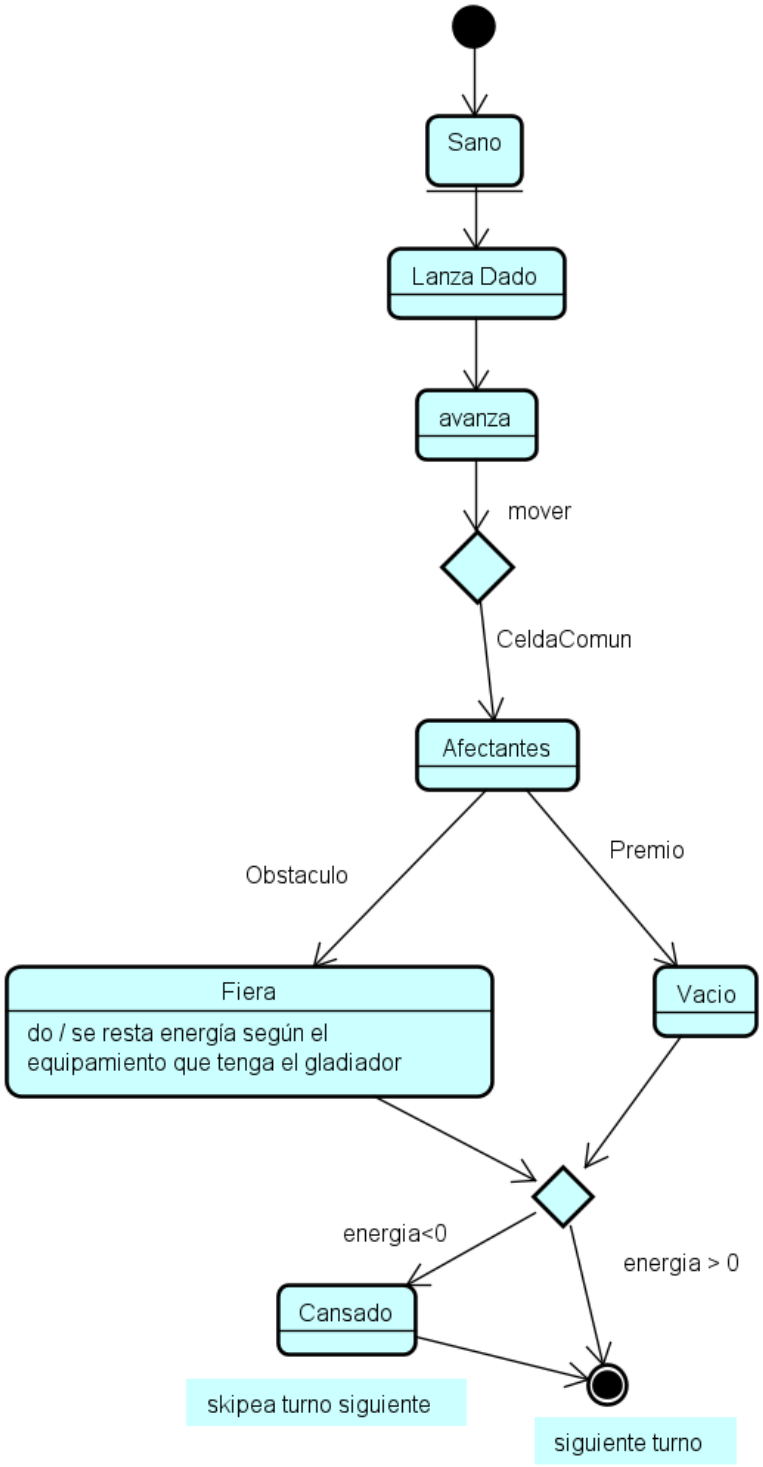

Figura 17: Jugar turno caso Fiera y Comida

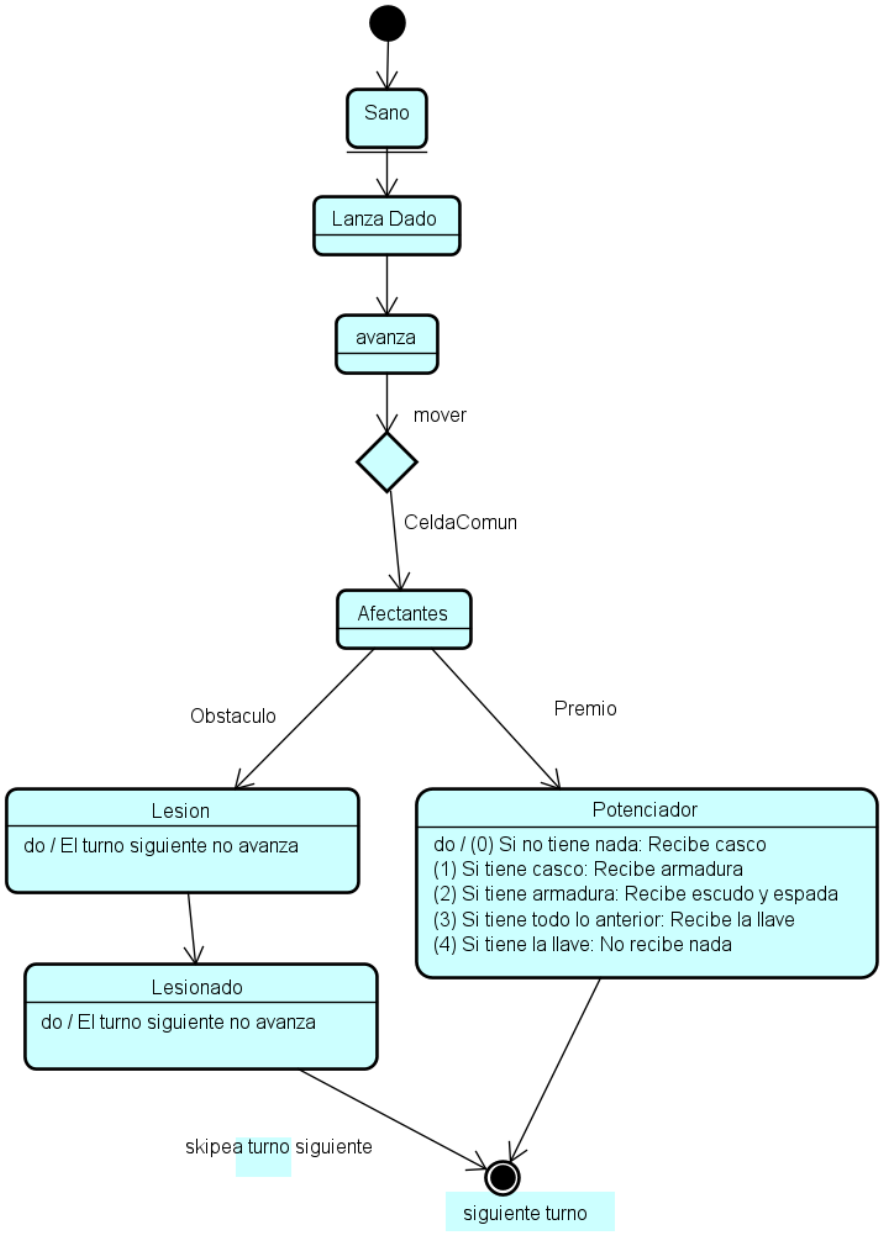

Figura 18: Jugar turno caso Lesionado

4. Detalles de implementación

4.1. Herencia

Decidimos aplicar la herencia al modelar las celdas, el equipamiento (ver figuras 3, 4 y de la sección Diagramas de Clase) y la clase Observador ya que en todos estos casos creemos conveniente y necesaria la reutilización de código. Esto es asi ya que las clases hijas presentan comportamiento en común.

4.2. Delegación

Hemos utilizado la delegación de forma transversal en todo el modelo. Creemos que es indispensable para no violar el principio de tell don´t ask y dividir de forma eficiente las responsabilidades de cada clase. A modo de un claro ejemplo, en la clase Gladiador se evidencia el uso de este pilar al delegar al estado del mismo la responsabilidad de mover efectivamente al gladiador a una nueva celda.

4.3. Patrón MVC

El uso del patrón MVC nos permite, además de organizar estructuralmente el proyecto, lograr vincular la lógica de nuestro juego (esto es, los requisitos y el desarrollo básico de nuestra aplicación) con la UI (User Interface - Interfaz Gráfica). Esto se logra al ligar las tres partes basicas que reconoce este patrón de diseño, los cuales son: Modelo (M), Vista (V) y Controlador (C). (ya que es un patrón de arquitectura). Aquí, una vista es un conjunto de elementos a través de los cuales el usuario interactúa con nuestra aplicación, ya sea que lo haga de forma activa o pasiva. Con el objetivo de visualizar mejor esta idea, un ejemplo de los elementos que pueden introducirse en un vista pueden ser: imágenes, sonidos, botones, etiquetas, etc. Por otro lado, la responsabilidad del controlador es extraer, modificar y proporcionar datos al usuario. Esencialmente, el controlador es el enlace entre las vistas y el modelo. La manera en que implementamos este patrón en nuestra aplicación tiene una estrecha relación con el patrón observer, explicado dos secciones adelante.

4.4. Patrón Double Dispatch

Hemos aplicado el double dispatch al modelar una parte del comportamiento de los Equipamientos, mas específicamente respecto a la mejora del mismo. De esta manera creamos la clase Potenciador, la cuál, a través de una sobrecarga del método "mejorarEquipamiento", que recibe un equipamiento concreto, implementa dicho comportamiento de forma particular para cada clase.

4.5. Patrón Observer

Hemos definido una clase abstracta Observado que simplemente tiene un array de observadores y un método para agregar un observador al mismo. Los métodos de las notificaciones específicas se implementan en cada clase concreta que hereda de Observado. También modelamos la interfaz Observador, que define los métodos que los observadores deben implementar para recibir actualizaciones. La idea básica detrás del uso que le dimos a este patrón es que en el momento en que el usuario interactúa activamente con la vista, se dispara un evento que es manejado por un controlador en particular, el cuál envia una directiva (mensaje) al modelo para reaccionar acorde a la interacción del usuario con la UI. De esta manera, si el estado de aquellas partes del modelo que son observados cambia, entonces se envia una "notificacion.ª cada uno de los observadores del mismo. En nuestra implementación en concreto, este último (observador) corresponde a VistaJuego, quien al recibir la notificacion modifica los elementos de la UI de forma adecuada.

4.6. Patrón State

Utilizamos este patrón para modelar los distintos estados en los cuales se puede encontrar un Gladiador en un momento particular del juego. Estos estados corresponden a: Sano, Lesionado y Cansado.

5. Excepciones

AfectanteInvalidoError : Se desencadena cuando el JSON necesario para crear las casillas de los afectantes no es válido.

ArchivoNoEncontradoError : Se genera cuando no se encuentra el JSON necesario para construir el tablero.

CantidadInvalidaDeCeldasError : Se arroja si el JSON del mapa contiene menos de 2 casillas.

CoordenadaInvalidaError : Se produce cuando el JSON del mapa incluye casillas con coordenadas negativas.

PasaronTreintaRondasYnoHuboGanadorError : Se activa al llegar al final del juego sin que haya un ganador después de treinta rondas.

TipoDeCeldaEnArchivoNoValidaError : Se desencadena si el JSON del mapa ontiene una casilla que no sea del tipo válido, como salida, camino o llegada.

UnJugadorGanoLaPartidaError : Se lanza cuando un jugador equipado con la llave llega a la casilla final, declarándolo como ganador.