FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — AlgoRoma

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

Alumno	Número de padrón	Email
Luciano Noel Wilberger Schwaab	107137	lwilberger@fi.uba.ar
Pablo Gaston Choconi	106388	pchoconi@fi.uba.ar
Gerónimo Paulozzi Molina	102340	gpaulozzi@fi.uba.ar
Mariano Barrionuevo	109629	mbarrionuevof@fi.uba.ar
Zoilo Pazos	107740	zpazos@fi.uba.ar

Índice

1. Introducción

Este informe reúne la documentación de la solución del segundo trabajo práctico de la asignatura Algoritmos y Programación III, que consiste en desarrollar un juego llamado AlgoRoma, que consiste en varios gladiadores se dirigen desde el Coliseo de Roma hasta una casa de Pompeya. Para poder ingresar tienen que obtener una llave mientras enfrentan desafíos durante su viaje. Este juego se puede jugar con un mínimo de 2 personas y un máximo de 6.

2. Suposiciones

1) Cuando la energía del gladiador es igual a 0, no puede moverse.
2) No hay límite de energía tanto en forma positiva como negativa.
3) Cuando un gladiador llega a una casilla, primero le afecta el premio y luego el obstáculo.
4) La energía del rango se obtiene al principio del turno.
5) Cuando un gladiador pierde un turno por no tener energía, no se toma en cuenta la energía ganada por el rango
6) En el caso de que los gladiadores vuelvan al medio del mapa por no tener la llave, no se afectan con los premios y obstáculos del casillero.
7) Los nombres de todos los gladiadores de una partida deben ser diferentes.

3. Modelo de dominio

El modelo consiste en tres clases principales que son Game, Gladiator y Square. Game tiene una lista de Gladiators y una de Squares y es el intermediario entre ambas. Se encarga de que los gladiadores jueguen su turno, de decidir si se termina el juego o sigue activo, y de afectar a los gladiadores con los premios y obstáculos de los casilleros. Los gladiadores tienen distintos comportamientos a la hora de jugar su turno, dependiendo de su estado y del premio y/o obstáculo que recibe. Es el encargado de actualizar su rango y de manejar su energía con respecto a los mensajes que recibe de su entorno. Por su parte, Square se encarga de tener los efectos de premio y obstáculo. Tanto Gladiator como Square tienen una Position, la cual va a ser fundamental para el desarrollo del juego. Para el desarrollo de la interfaz gráfica utilizamos el patrón de arquitectura MVC en los que la vista y el modelo no interactúan entre sí, sino que hay un controlador que se encarga de relacionarse con ambas.

4. Diagramas de clase

En este diagrama se puede ver la clase Game y sus interacciones con las otras clases. Contiene dos o mas gladiadores y un conjunto de Squares al que lo consideramos el camino a recorrer. Tiene además un atributo que implementa la interfaz GameState, la cual realiza distintos comportamientos analizados en el segundo diagrama(ver figura 2)

Figura 1: Diagrama de la clase Juego.

La interfaz GameState tiene 3 implementaciones, las cuales realizan los métodos de diferentes maneras. ActiveGame instancia otros estados de juego, cuando al finalizar un turno se cumplen las condiciones para que el juego termine, ya sea por cantidad de turnos jugados o por si un gladiador llega a Pompeya con la llave.

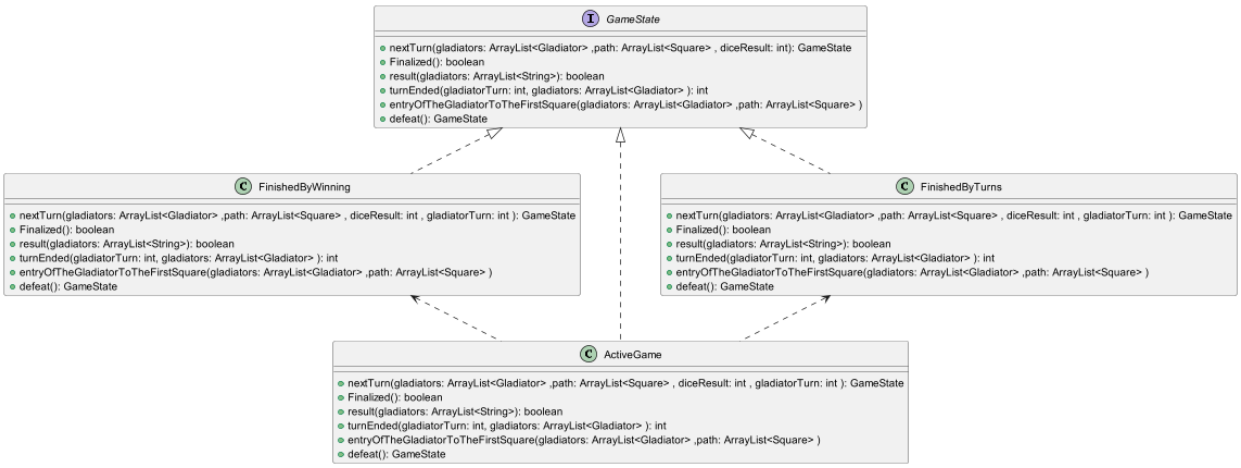


Figura 2: Diagrama de los estados de juego.

Gladiador tiene como atributo un rango Rank que lo ayuda a conseguir energía conforme vallan pasando los turnos y un equipamiento Equipment que lo consigue en el camino y le sirve para ganar la partida o también para perder menos energía enfrentando a las bestias. Por otro lado, tiene un estado State, que define su comportamiento ya que delega en él la mayoría de sus métodos

(ver figura 4).

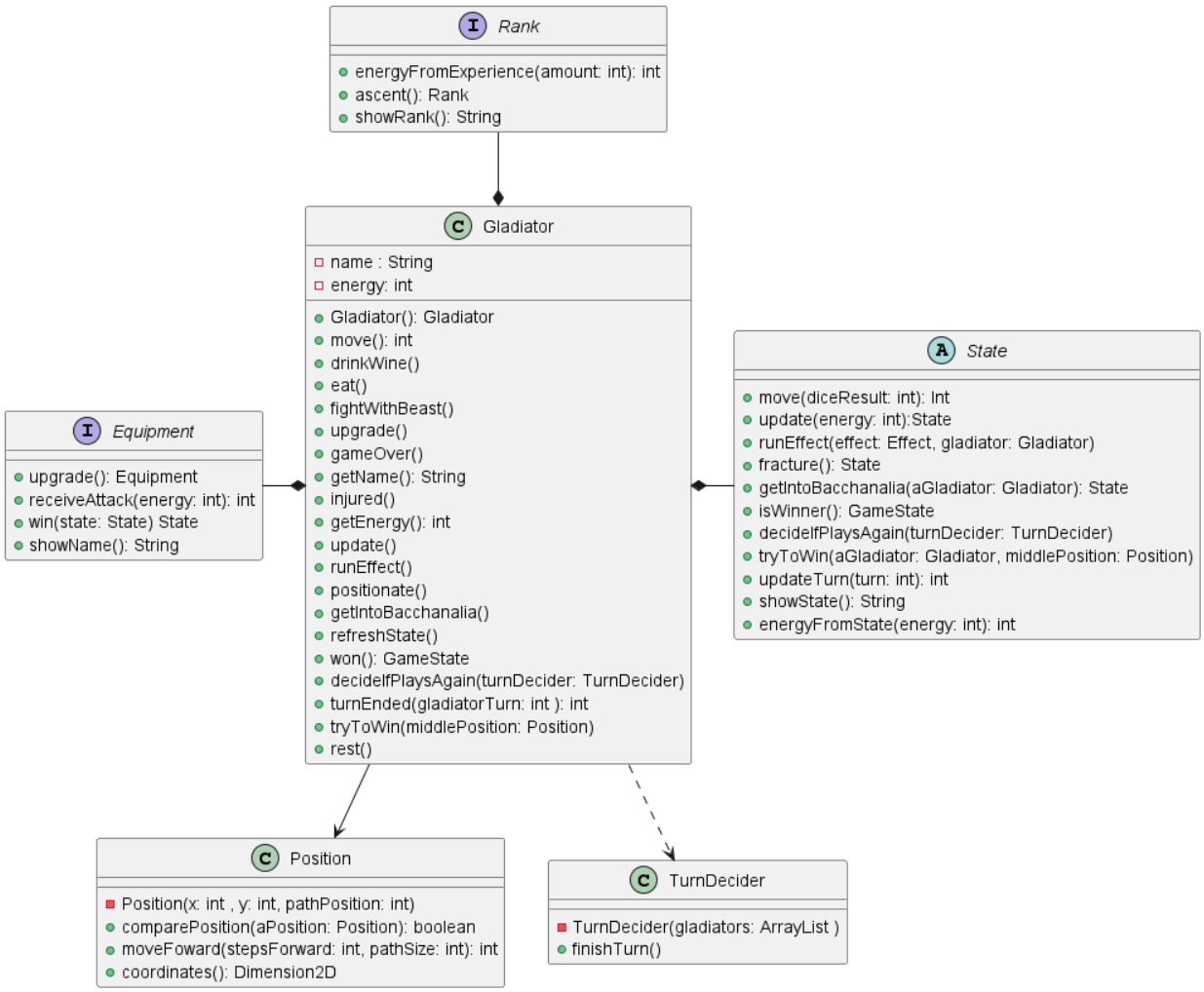


Figura 3: Diagrama de Gladiator.

El State de Gladiator tiene 5 clases hijas, y todas tienen implementaciones específicas. State se encarga de decidir si se mueve en un turno, si se le aplica o no los efectos recibidos y decide si finaliza el turno del gladiador. Por otro lado define si el gladiador ganó el juego o no. En caso de no tener energía, se instancia Tired.En caso de caer sobre un casillero de lesión, se instancia Injured; lo mismo sucede al caer en un Bacanal, se instancia InTheBachanallia.

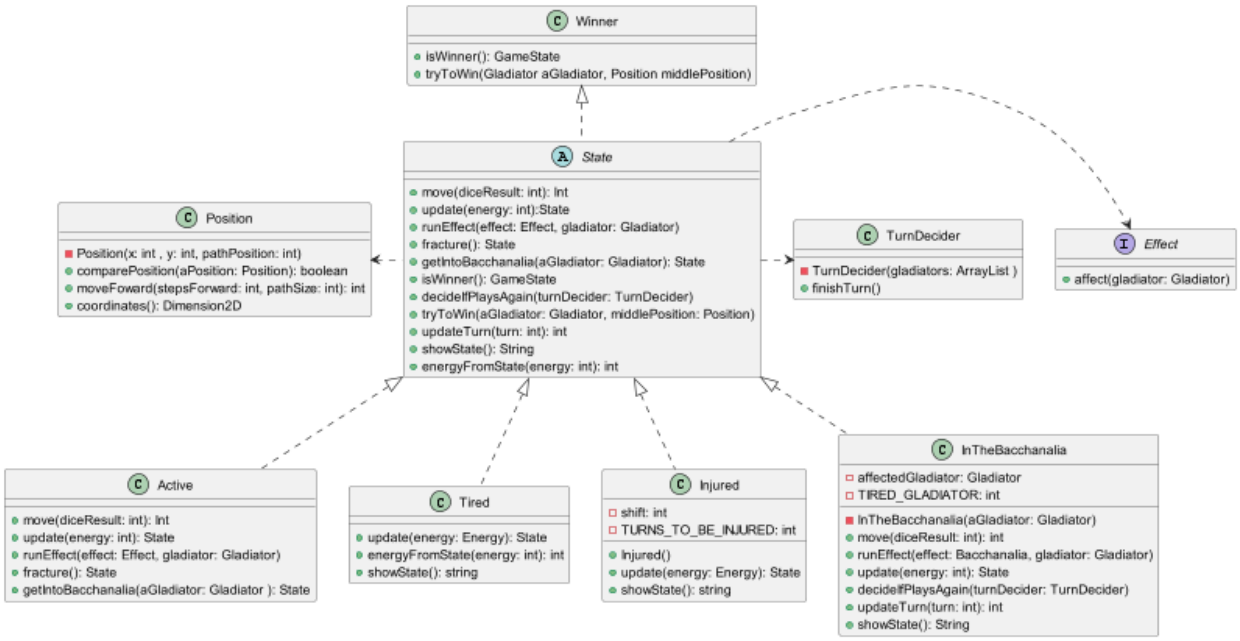


Figura 4: Diagrama de los estados de Gladiator.

La clase Square tiene la finalidad de:

1. hacer que al gladiador se le efectué los efectos que este posea ya sean 1, 2 o ninguno
2. hacerle saber al gladiador donde esta posicionado para que pueda avanzar a los casilleros siguientes de manera ordenada (como esta hecho el mapa).

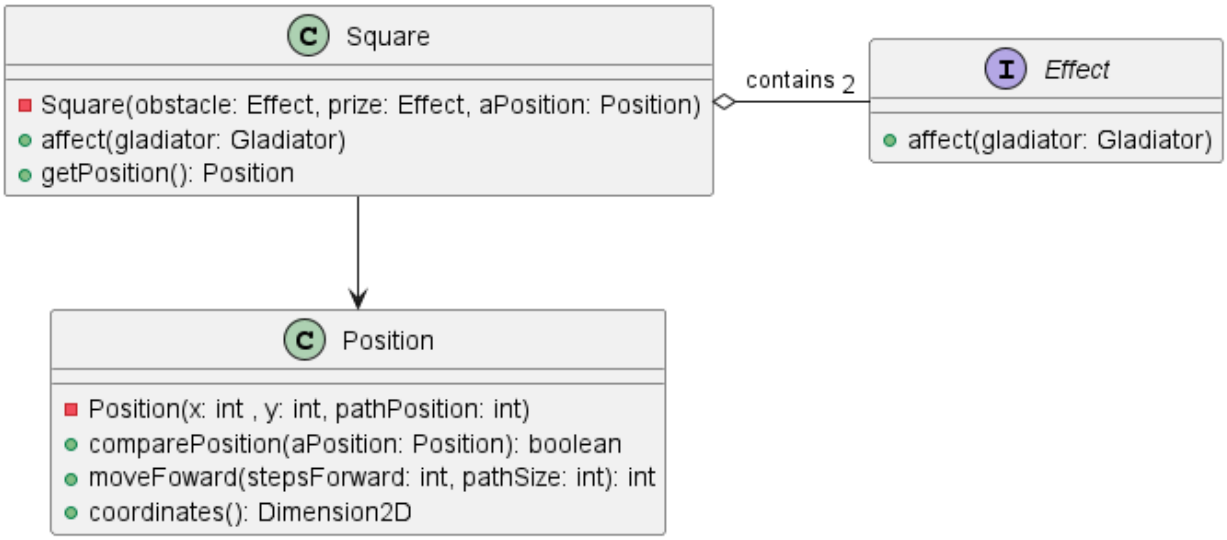


Figura 5: Diagrama de los squares.

5. Detalles de implementación

5.1. Patrones de diseño

5.1.1. Patrón State

Se aplica el patrón state en los equipamientos, rangos y estados del gladiador, para modificar su comportamiento dependiendo de cuales tiene. El siguiente diagrama representa el patrón aplicado para el equipamiento.

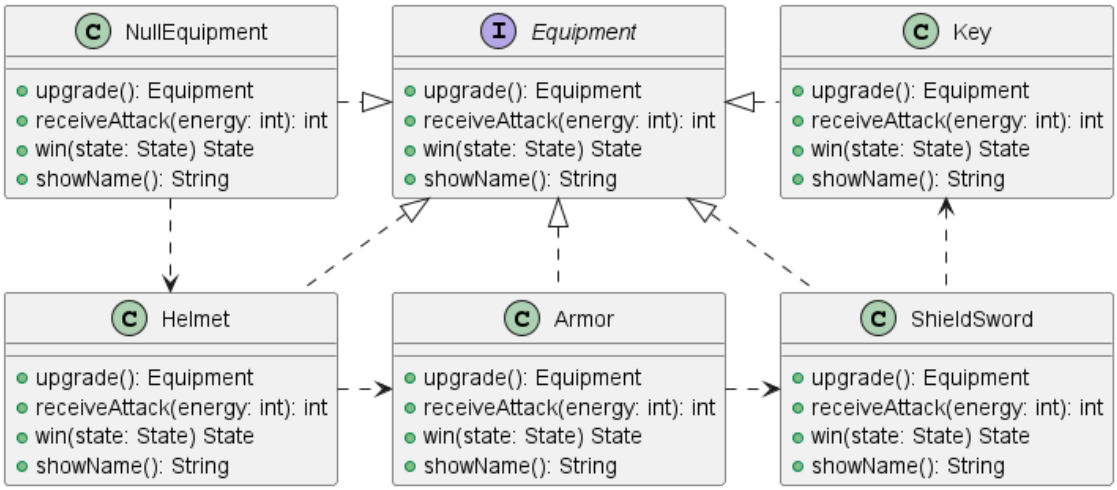


Figura 6: Diagrama de clase Equipment.

5.1.2. Patrón Observer

Se aplica el patrón observer para actualizar la posición de los gladiadores en el mapa. Su energía, equipamiento y rango en la barra lateral. También se utilizar para mostrar el turno actual del juego.

5.1.3. Patrón NullObject

Se aplica el patrón para lidiar con las casillas sin efectos, haciendo uso de una clase NullEffect. También se utiliza para un gladiador sin equipamiento. De esta forma podemos

5.1.4. Patrón Facade

Se aplica el patrón Facade para ocultar la implementación del parseo del mapa. Estos nos permitiría poder agregar nuevos parsers para otros tipos de archivos además del .json.

5.1.5. Patrón Simple Factory

Para el caso de los efectos, que son los premios y obstáculos que afectan a los gladiadores, usamos el patrón de diseño de "Simple factory"que nos beneficio porque:

- Permitió desacoplar la lógica de creación de efectos del parser, a la hora de implementar un nuevo efecto simplemente habría que sumar un caso a EffectFactory.
- Centralizar la creación de efectos en una única clase, los cuales deben cumplir una interfaz común.

- Ayuda a aplicar el Principio de Responsabilidad Única al separar la responsabilidad de creación de objetos en una clase dedicada.

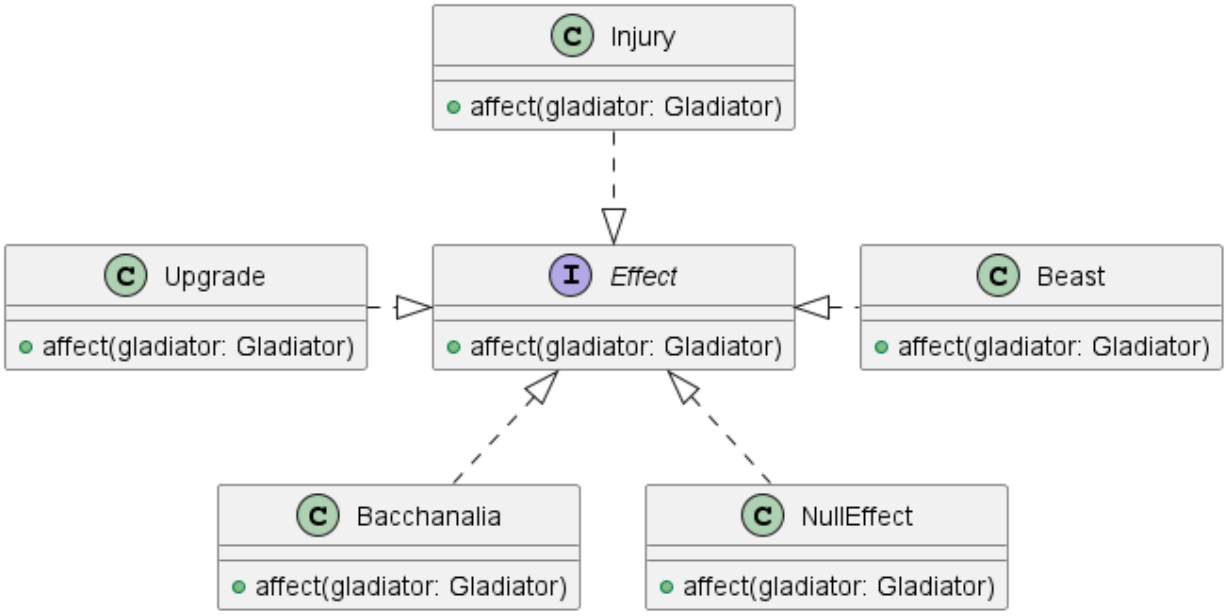


Figura 7: Diagrama de clase Effects.

5.1.6. Singleton

Se aplica el patrón para obtener acceso global de la única instancia de la clase, permitiendo:

1. Asegurarse de tener una única instancia.
2. Evitar pasar la instancia por parámetro.

Se aplica este patrón en:

- La clase general del juego (Game) ya que al ser solo necesaria una única instancia del mismo para los propósitos del programa, singleton nos permite acceder a la misma de forma global, conservando sus atributos.
- Messenger (clase que utiliza un logger pasado por parámetro), utilizar el patrón singleton con loggers tiene beneficios como utilizar una sola configuración del mismo y asegurar un formato consistente en todos los mensajes a los mismos destinos, tener varias instancias de loggers podría resultar en mensajes con distintos formatos.
- El controlador de sonido (Sound) ya que todos los sonidos son cargados al inicio de la ejecución, por lo cual resulta conveniente acceder siempre a esa misma instancia.

5.2. Principios y pilares

5.2.1. Herencia

Se aplico herencia en el estado de los gladiadores (Clase State), lo cual permite evitar código repetido, ya que hay varias clases hijas que mantienen el mismo comportamiento.

Se evito utilizar herencia en los efectos de las casillas ya que al momento de la implementación del modelo no se distinguió ningún beneficio que justificara el aumento de acoplamiento. Aunque

luego de armar las vistas notamos que si se puede beneficiar ya que como se utiliza el patrón observer permitiría ahorrar escribir múltiples veces los métodos para el observer.

5.2.2. Principio de inversión de dependencia

La clase Square no conoce los efectos, solo conoce la interfaz Effect.

5.3. MVC

Se aplicó la arquitectura MVC durante el desarrollo de la aplicación, el modelo no conoce las vistas ni tampoco la librería gráfica utilizada. Aunque quedaron pendientes algunos refactors para quitar referencias del modelo en algunas vistas.

6. Excepciones

InvalidMapFile Se lanza cuando el formato del archivo .json del mapa es invalido.

MapFileCouldNotBeParsed Se lanza cuando falla el parseo del archivo del map

MapFileNotFound Se lanza cuando el archivo del mapa no fue encontrado

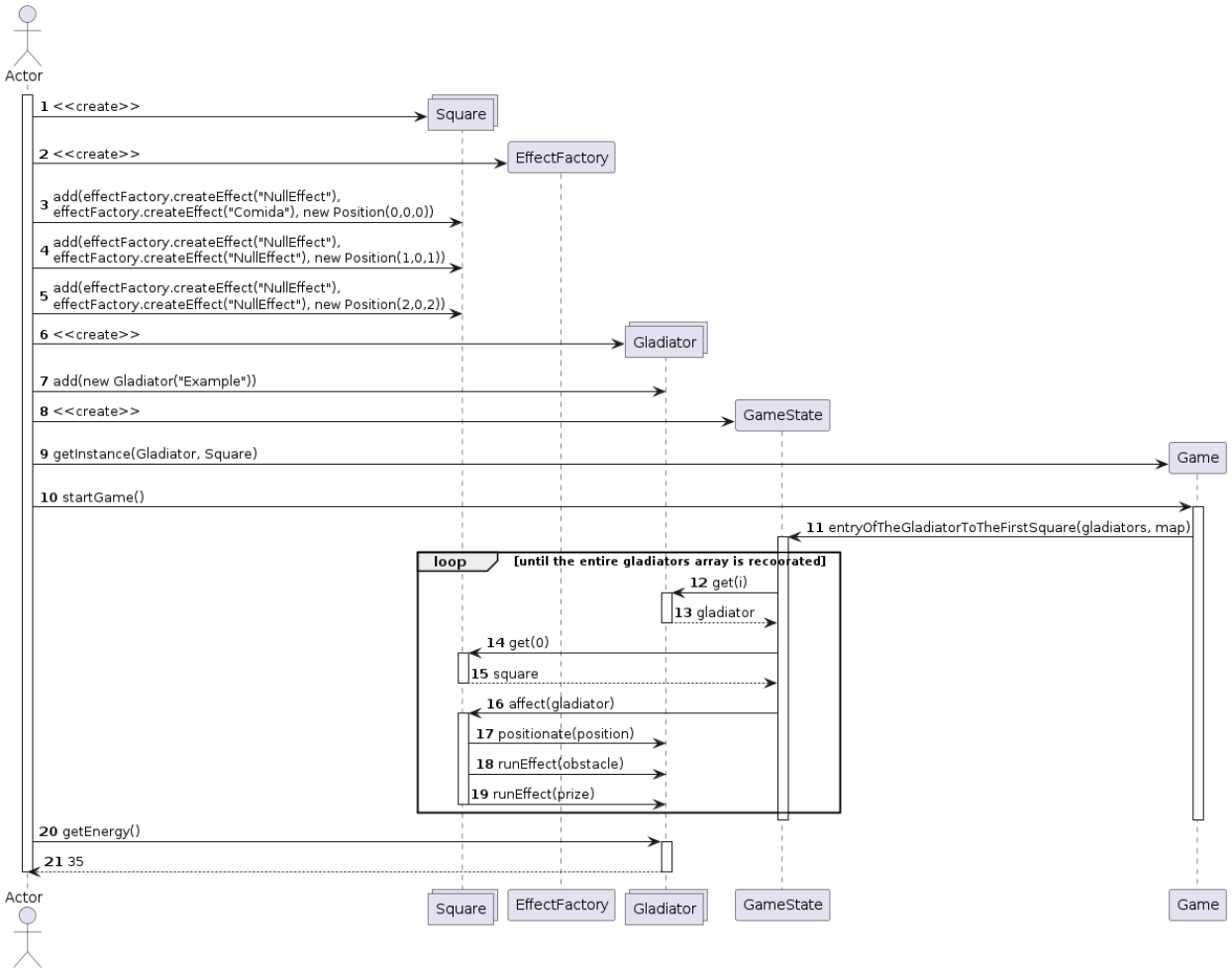
MapFileFailedToOpenOrClose Se lanza cuando el archivo del mapa no pudo ser abierto o cerrado. No esta testeada ya que no encontramos forma de hacer fallar el abrir/cerrar del archivo o mapa, aun así para el caso en el que esto pudiera llegar a suceder.

ErrorSoundNotFound Se lanza cuando falle la búsqueda de un archivo de audio para los sonidos del juego.

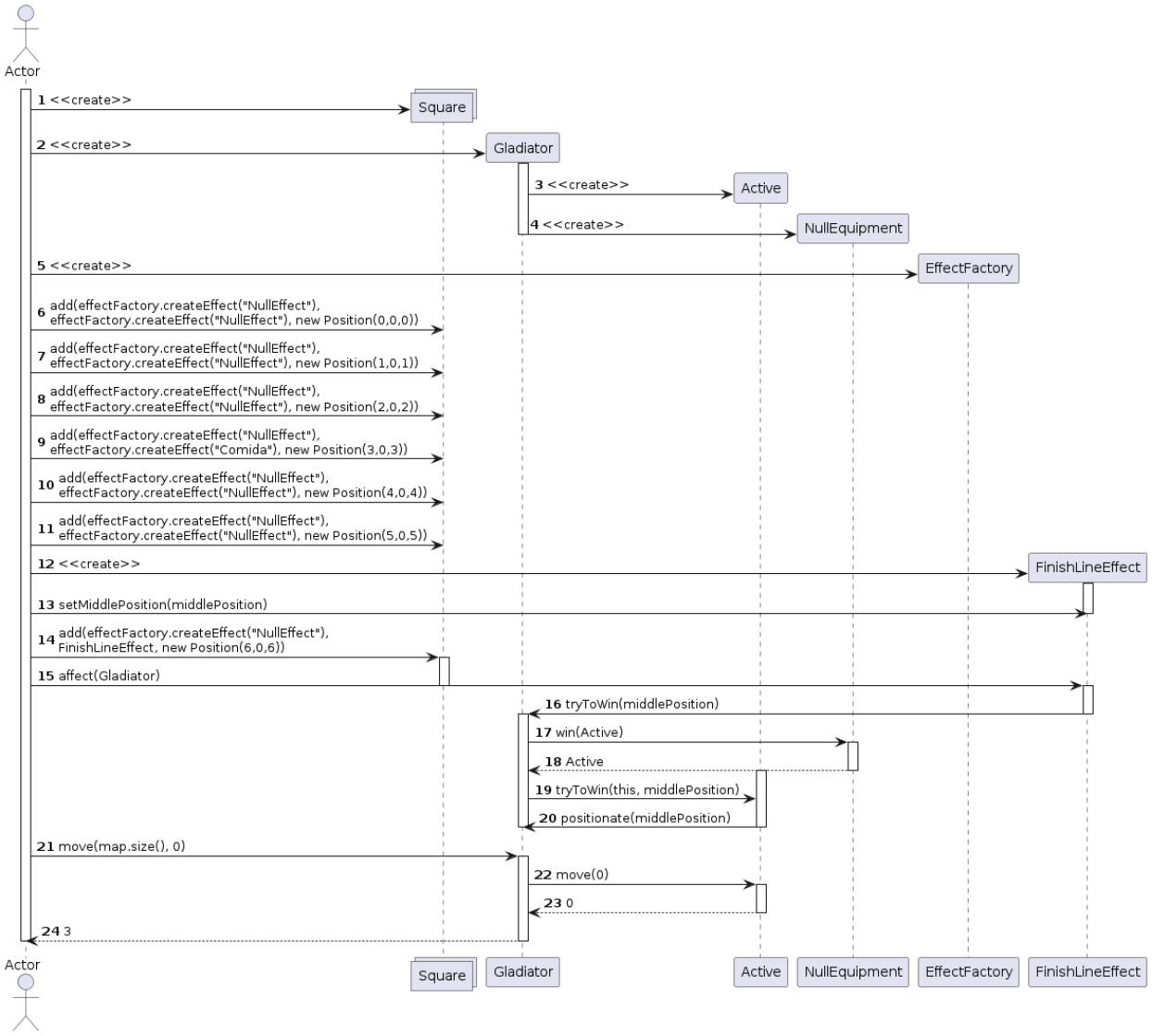
7. Diagramas de secuencia

7.1. Se incorpora un gladiador en el casillero inicial (Use case 2)

Este trata de la incorporación de un gladiador en el casillero inicial, en este caso el primer casillero tiene el efecto comida, de esta manera se puede comprobar que si se encuentra en la casilla inicial a los gladiadores se les deberá sumar 15 puntos de energía.

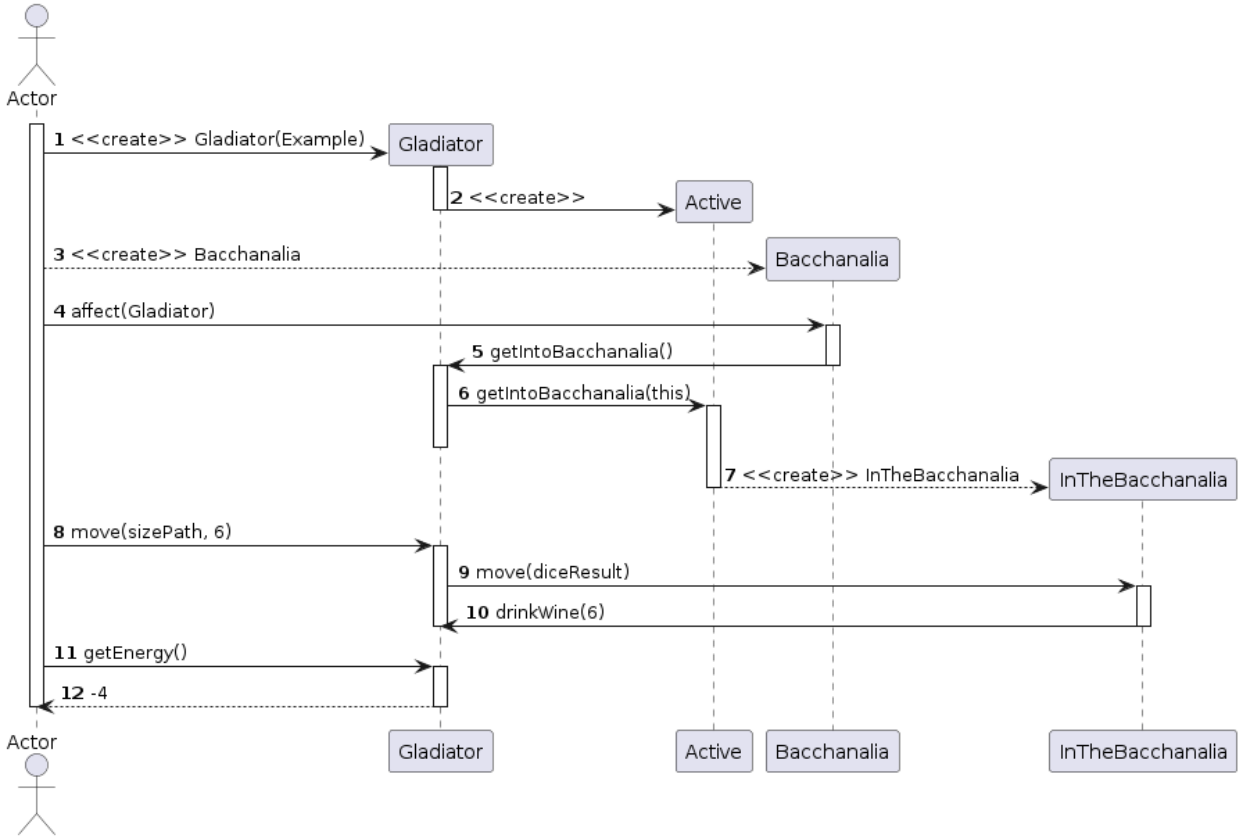


7.2. Efecto del casillero final cuando un gladiador no tiene llave

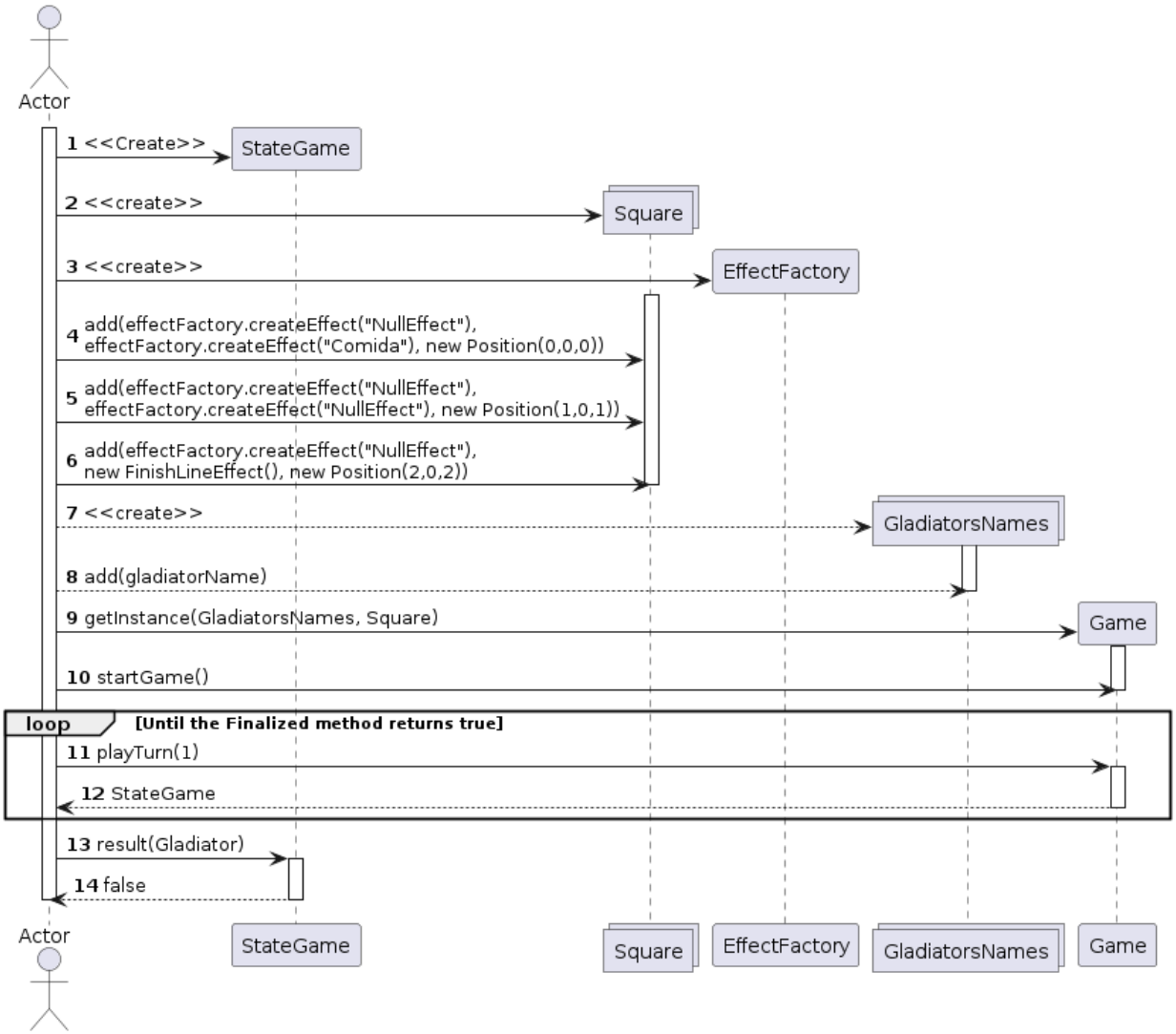


7.3. Gladiador sufre el efecto Bacanal

Acá se muestra lo que sucede cuando se efectúa todo el sistema del efecto bacanal



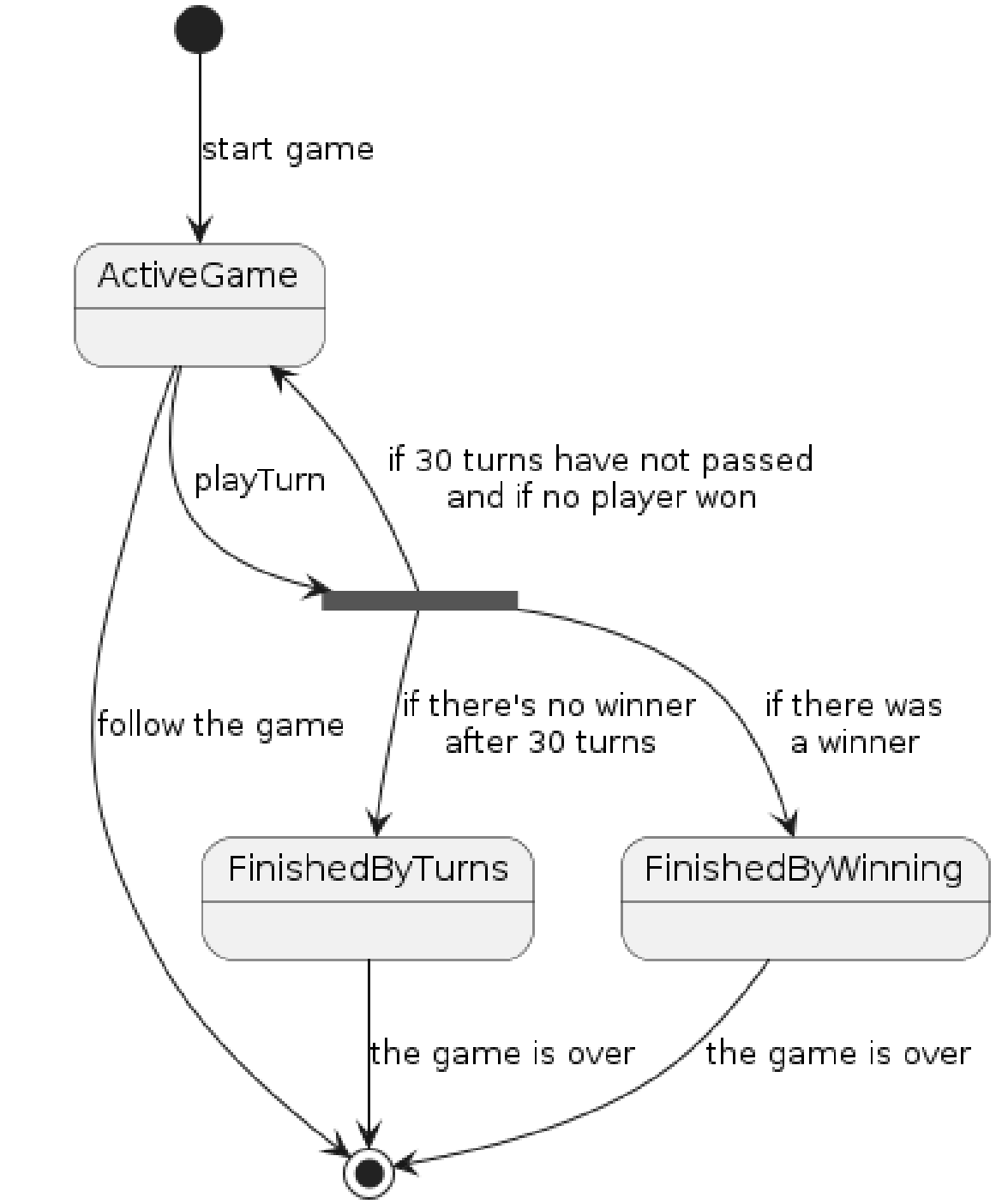
7.4. Sistema del juego cuando pasan los 30 turnos



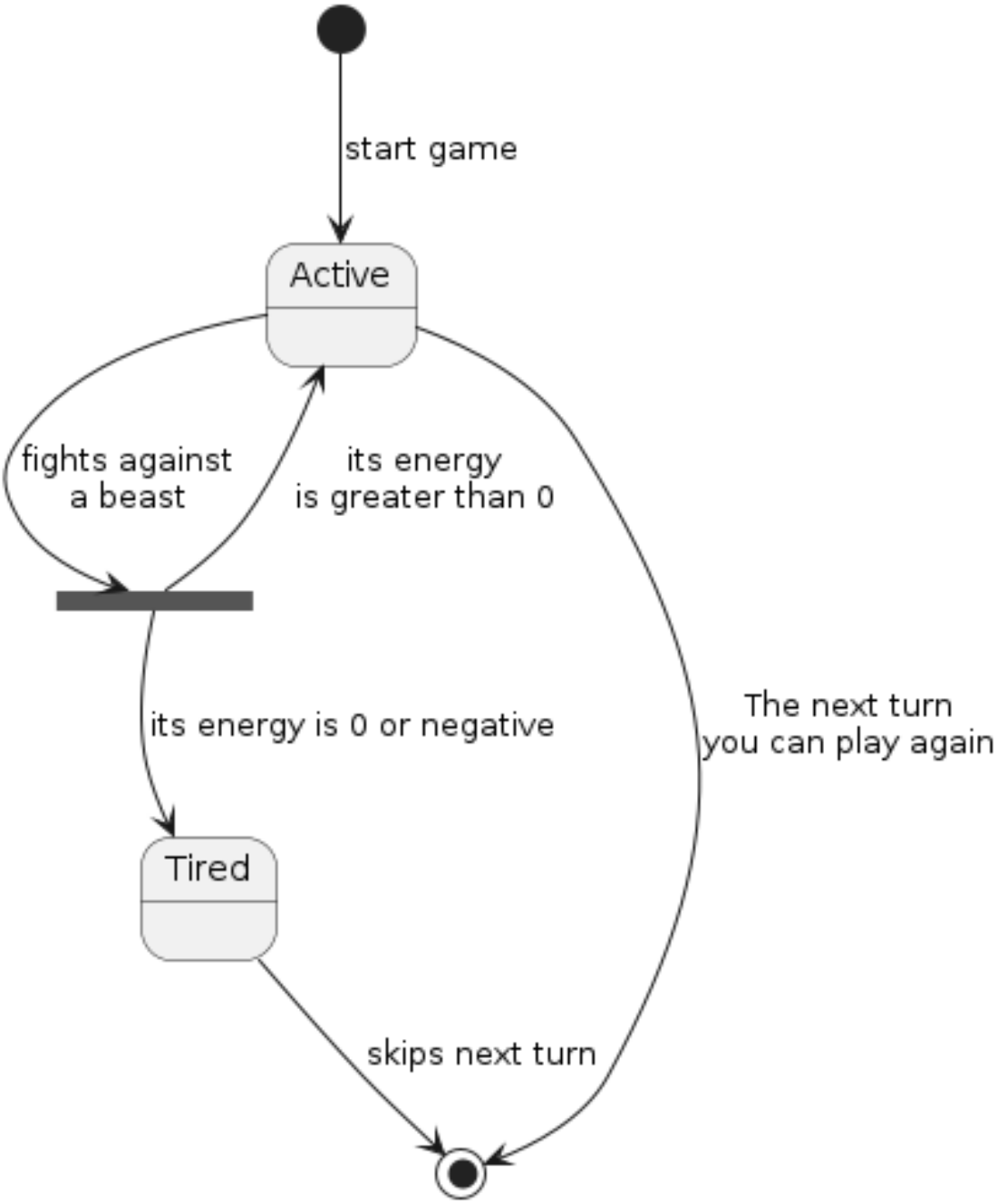
8. Diagramas de Estado

8.1. Estados de juego

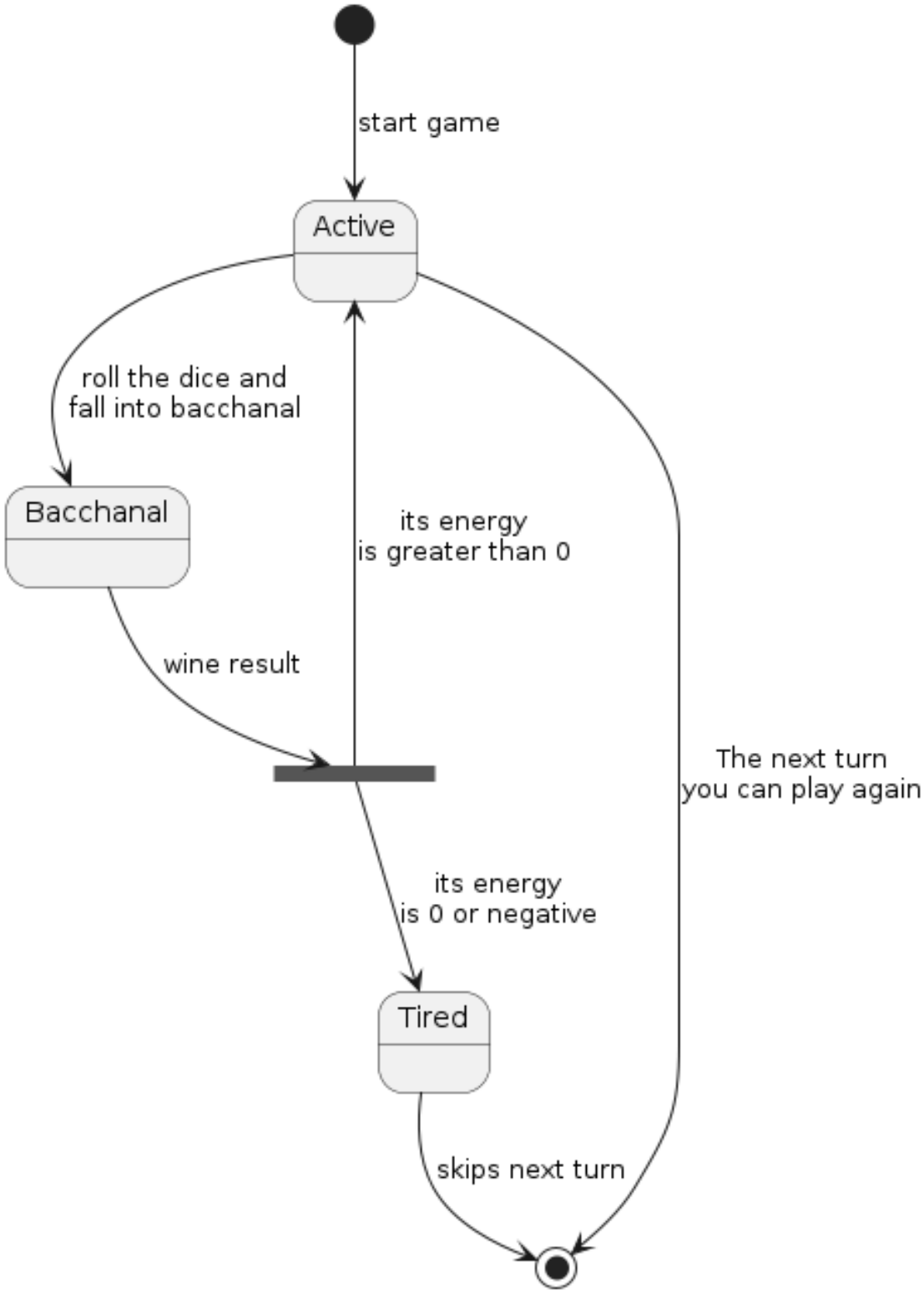
Acá se va mostrando los estados que puede ir teniendo el juego a lo largo de la partida dependiendo de lo que suceda en ella



8.2. Estados posibles de Gladiador al enfrentar a una bestia



8.3. Estados posibles de Gladiador al entrar a un Bacanal



9. Diagramas de paquete

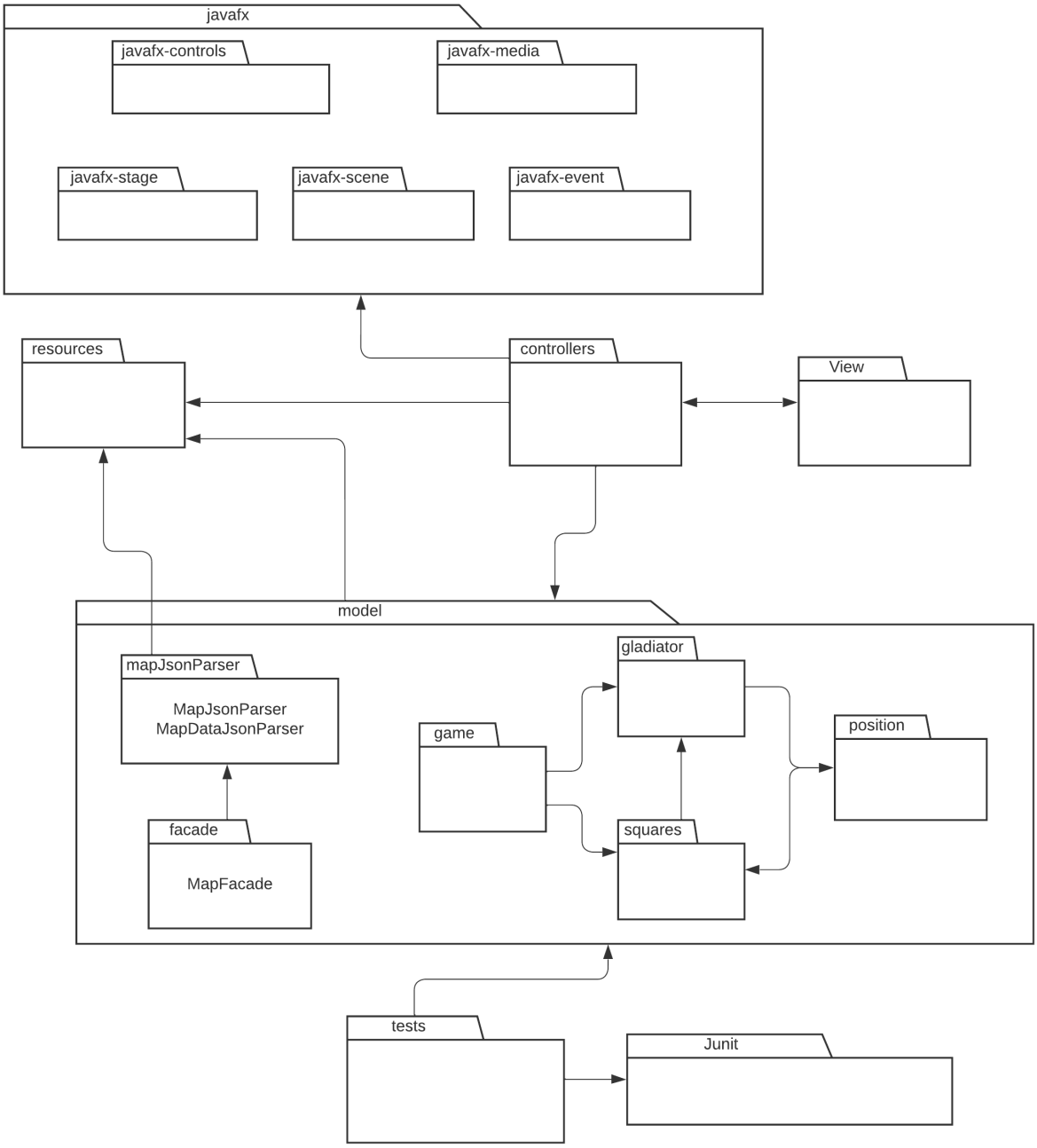


Figura 8: Sistema del Programa