FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico Nro.: 2

AlgoRoma

Edición: "Gladiadores en Fuga "

[7507/9502] Algoritmos y Programación III
Curso 2
Segundo cuatrimestre de 2023

Grupo	Nombre	Padrón	Email
7	Condori Zeballos, Favio Gustavo	101906	fcondori@fi.uba.ar
	Maldonado Barbosa, Henry Mauricio	99570	hmaldonado@fi.uba.ar
	Rodriguez, Nazareno Jose Luis	101891	nrodriguez@fi.uba.ar
	Servin Velazquez, Marcos	101855	maservin@fi.uba.ar
	Sosa, Juan Manuel	92810	jmsosa@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 que consiste en desarrollar una aplicación en Java de un juego de tablero, AlgoRoma, ambientado en el imperio romano utilizando los conceptos del paradigma de la orientación a objetos vistos en el curso.

2. Supuestos

- Al retroceder no recibe castigos ni premios del casillero
- Cada gladiador tiene un nombre único

3. Diagramas de clases

3.1. AlgoRoma

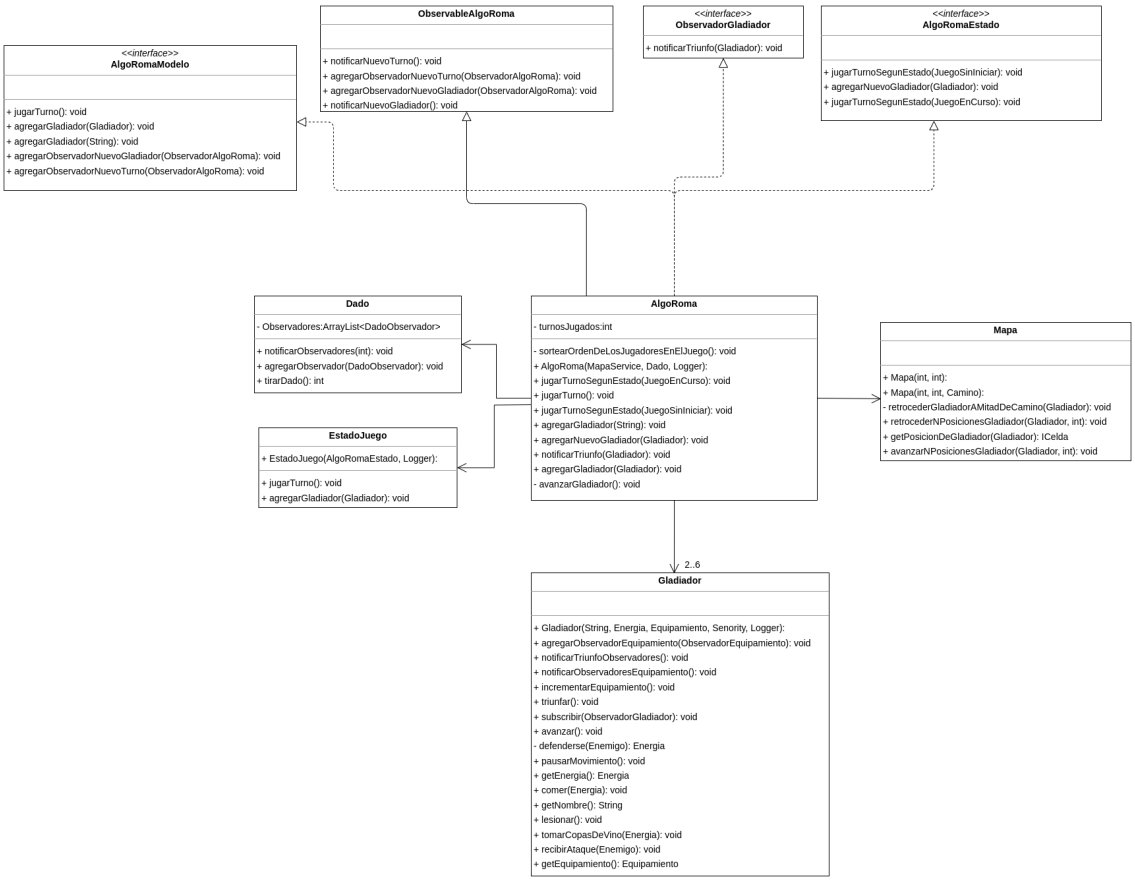

Figura 1: Diagrama de clase: AlgoRoma.

Representa el núcleo del videojuego , encapsulando un conjunto de dependencias esenciales para el desarrollo del juego, tales como el mapa, el estado de juego y el dado.

3.2. Mapa y sus dependencias

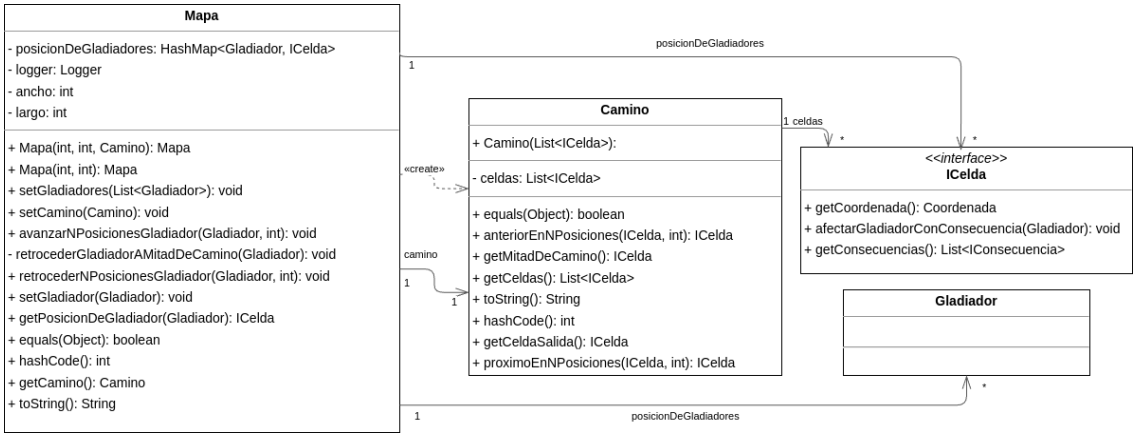

Figura 2: Diagrama de clase: Mapa.

En este modelo, las celdas son componentes basicos del mapa y el camino respresenta la ruta especifica a través de las celdas. El mapa actua como un contenedor para las celdas y proporciona el contexto para la mainipulación y navegación de los objetos. El enfoque adopdato brinda como principal ventaja la extensibilidad del diseño asi como la incorporación de nuevas funcionalidades.

3.3. Celda y sus dependencias

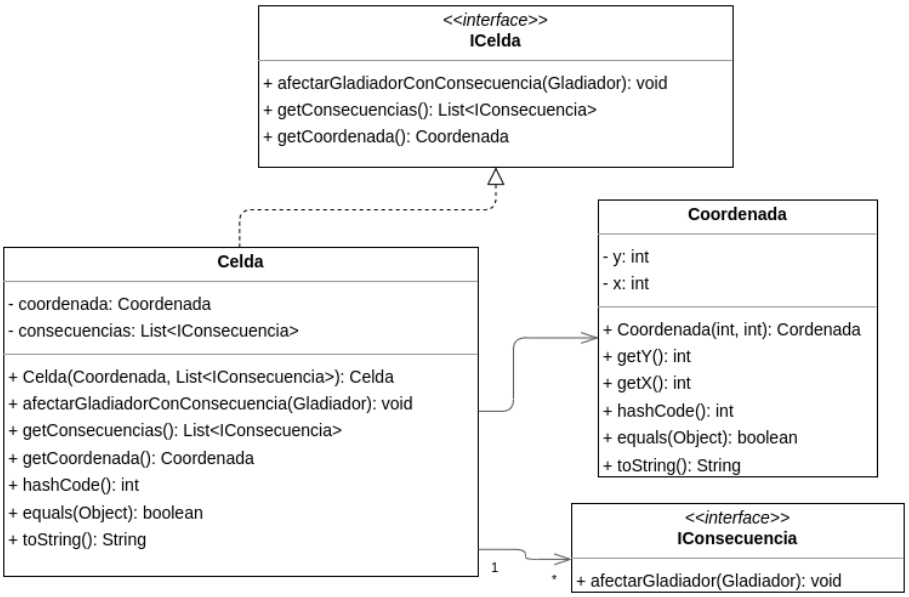

Figura 3: Diagrama de clases de la Celda.

La celda es la unidad elemental del mapa, representando un punto especifico con coordenadas(x,y). La interacción con otros objetos como por ejemplo gladiador, permite encapsular el comportamiento que este caso es la afectacion del gladiador.

3.4. Consecuencias

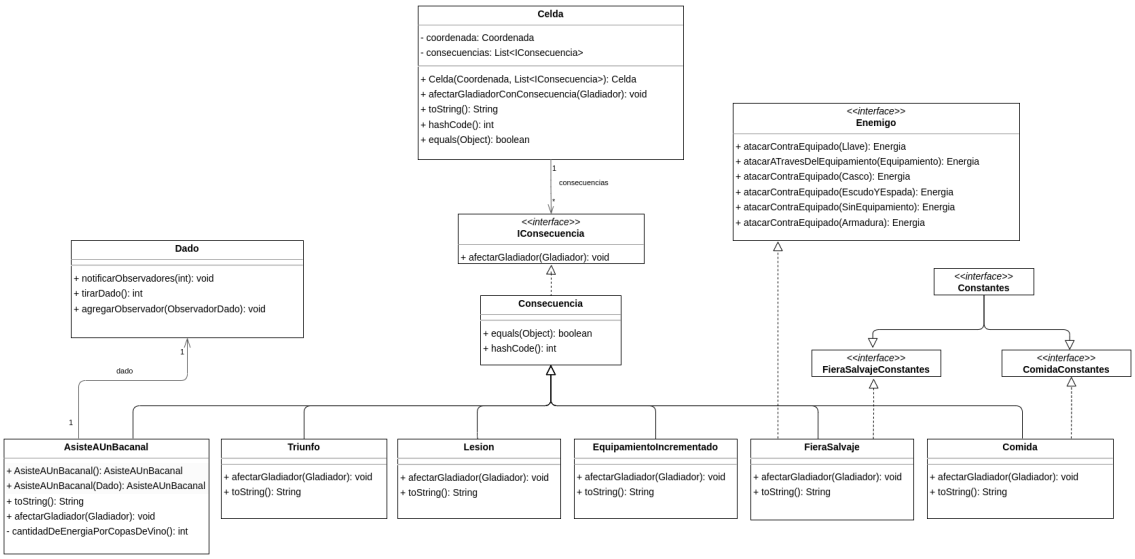

Figura 4: Diagrama de clase: Consecuencia.

Consecuencia define en el modelo del dominio todas aquellas interracciones que pueden cambiar de estado o valor de un atributo del gladiador, como por ejemplo: aumentar energía, disminuir energía, cambiar senority en otros comportamientos.

3.4.1. Enemigos

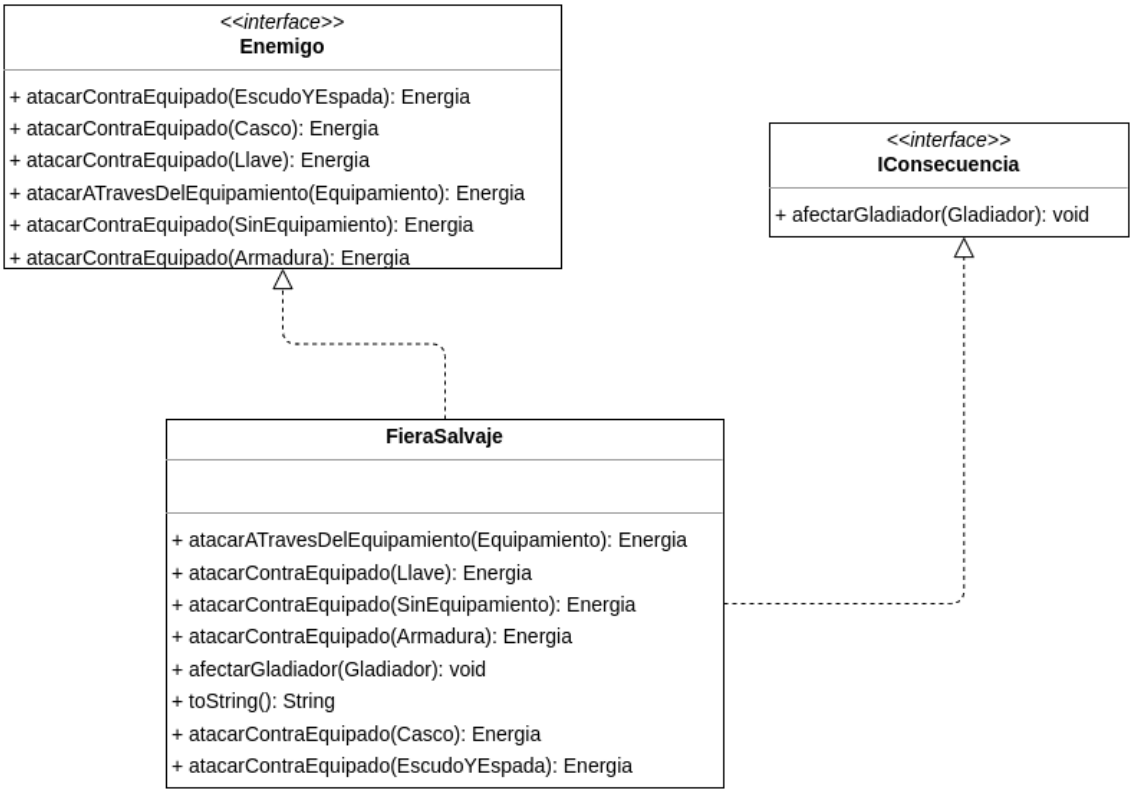

Figura 5: Diagrama de clase: FieraSalvaje.

Enemigo representa todo aquello que se puede enfrentar al gladiador. El tipo de daño se define a partir del equipamiento que puede tener el gladiador.

3.5. Gladiador y sus dependencias

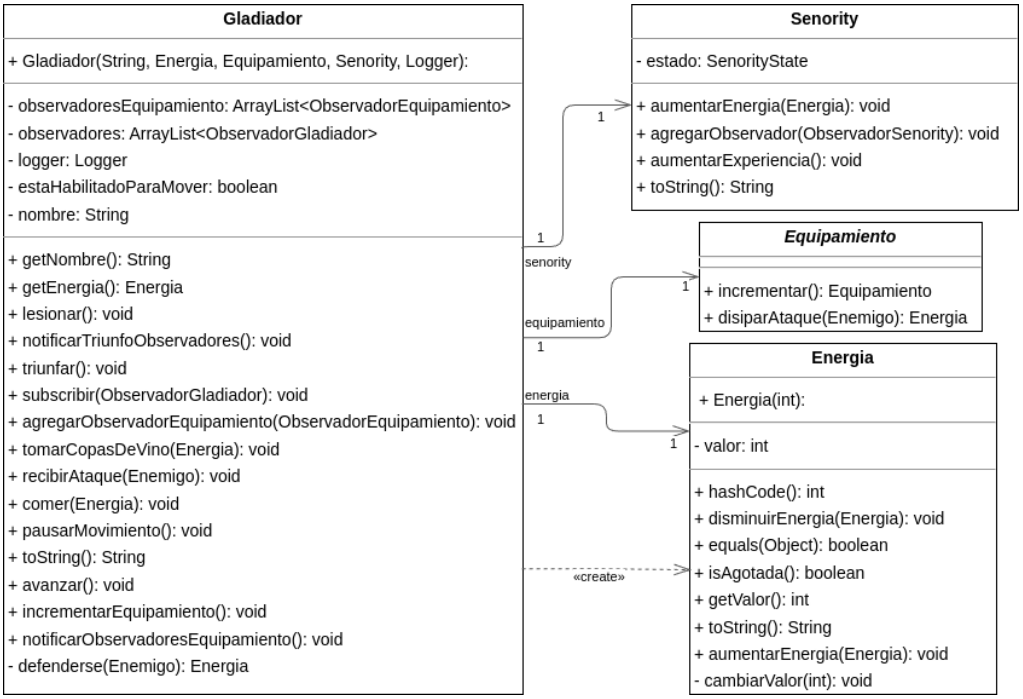

Figura 6: Diagrama de clases del Gladiador.

3.5.1. Equipamientos

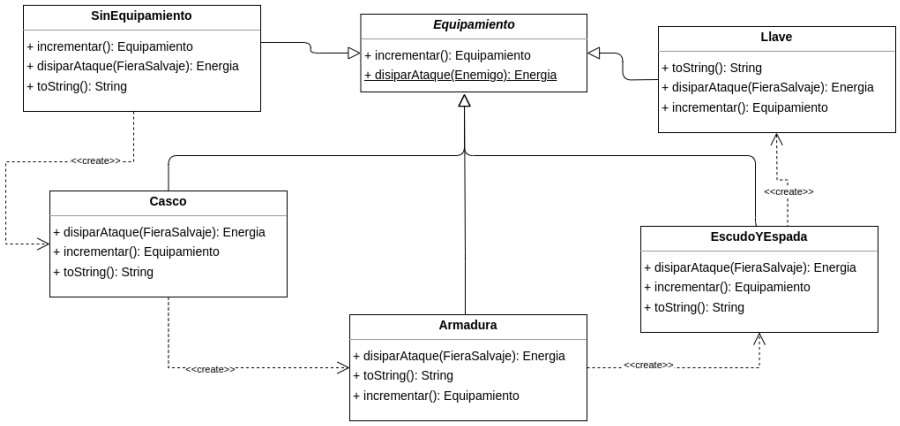

Figura 7: Diagrama de clase: Equipamiento.

Siendo que los equipamientos se incrementan de modo secuencial, a un Casco le continua una Armadura y a esta un EscudoYEspada, los mismo tiene la capacidad de incrementarse a través su método *incrementar():Equipamiento* que retorna el Equipamiento que le continua en jerarquía.

3.5.2. Senority

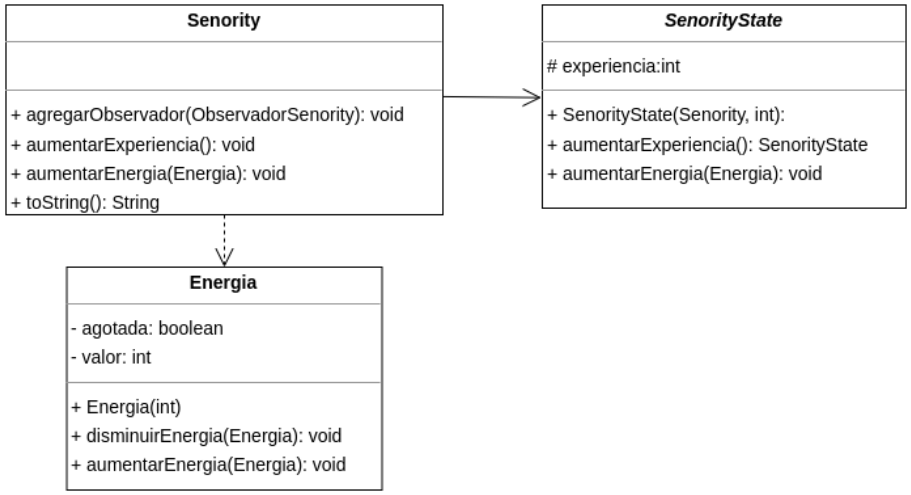

Figura 8: Diagrama de clase: Senority.

Representa aquella las comportamiento que puede realizar el gladiador,los cuales le permiten subir el senority o aumentar su energía.

3.5.3. SenorityState

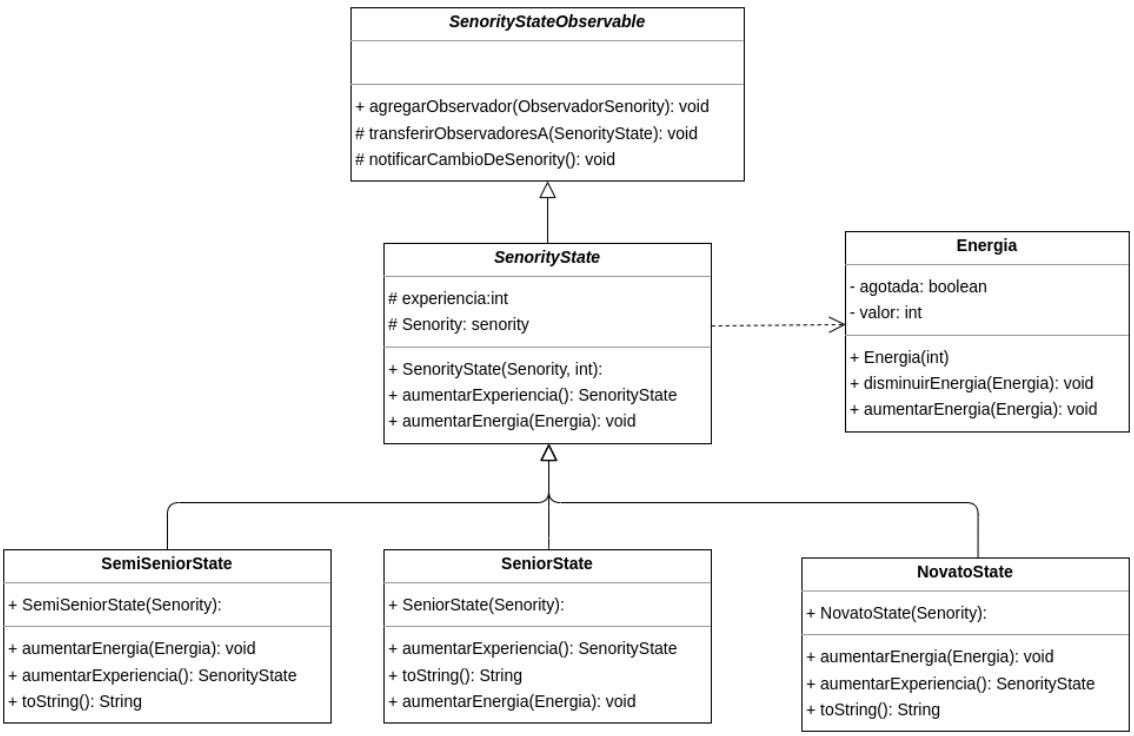

Figura 9: Diagrama de clase: SenorityState.

Reprsenta los estados de un gladiador y como esto influye en el comportamiento de la energía.

3.5.4. Energia

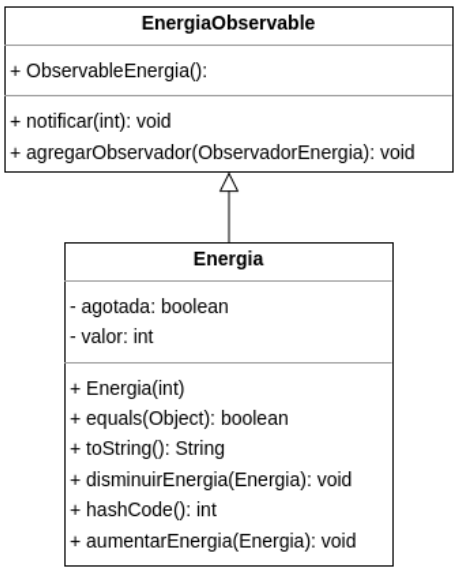

Figura 10: Diagrama de clase: Energia.

Reprsenta los puntos de energía de un gladiador.

3.6. MapaService y sus dependencias

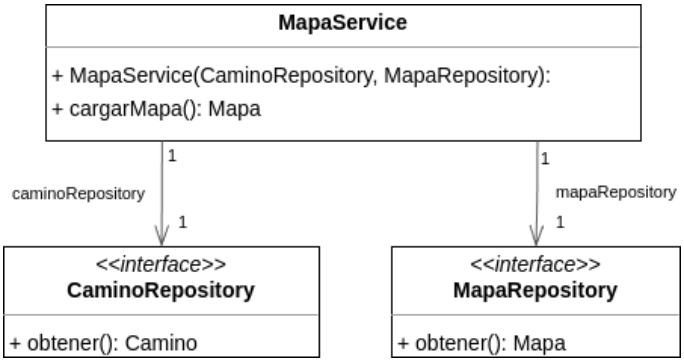

Figura 11: Diagrama de clase: MapaService.

3.6.1. Repositories

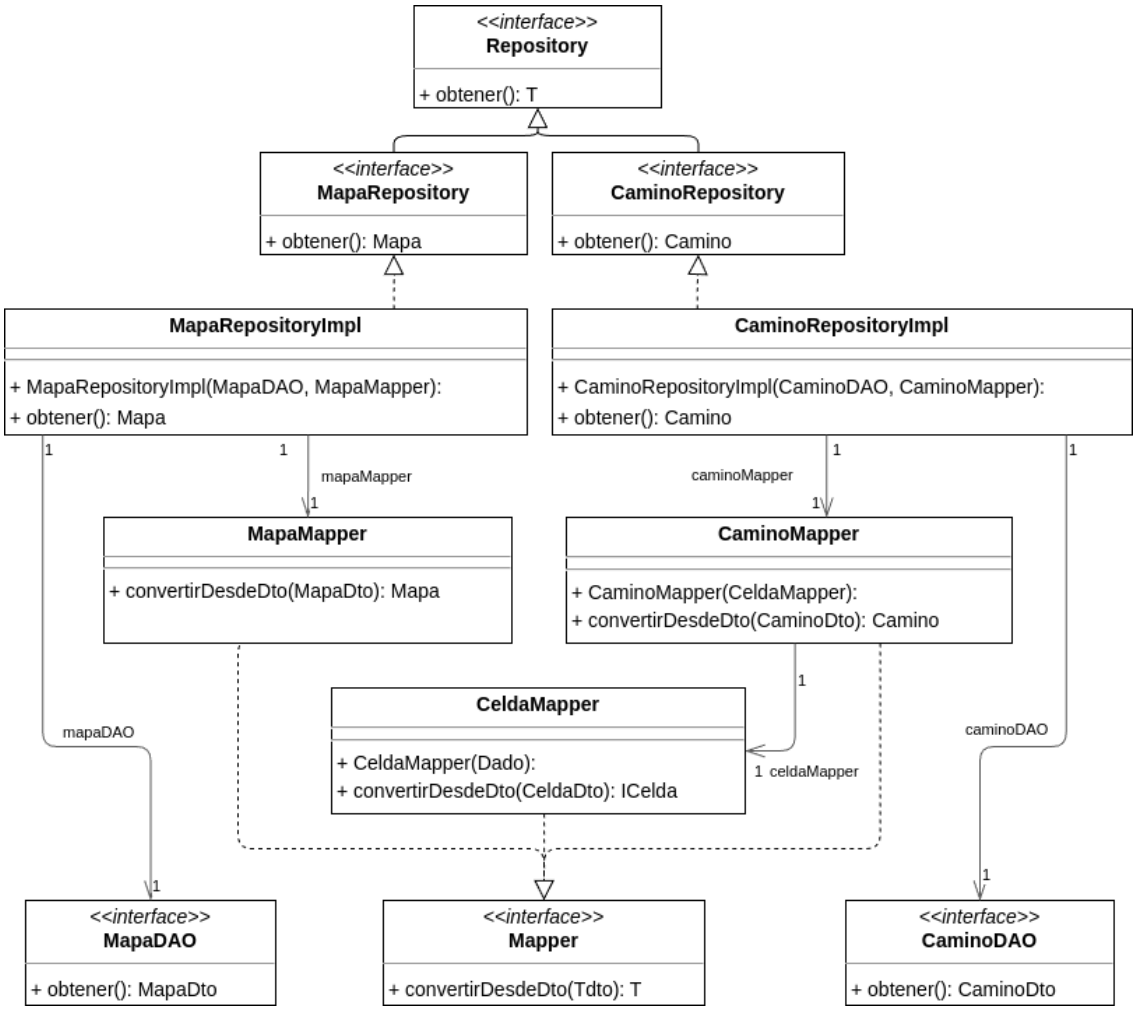

Figura 12: Diagrama de clase: Repositories utilizados por mapaService, con dependencia de Mappers y DAOs.

3.6.2. DAO

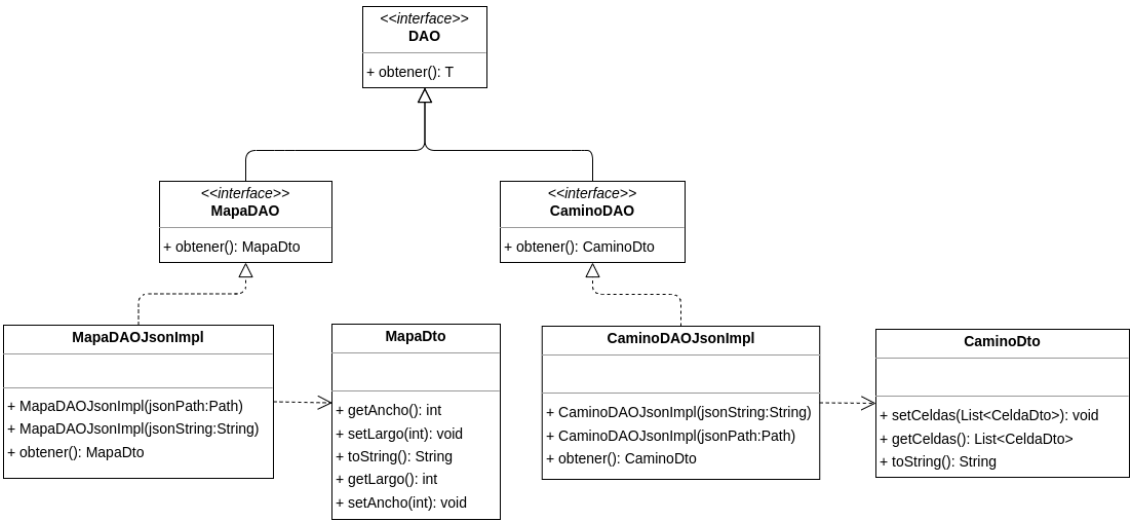

Figura 13: Diagrama de clase: DAO.

A partir de un archivo JSON, se lleva a cabo el proceso de deserialización con el fin de obtener la estructura de datos que conforma los objetos mapaDto y caminoDto. Este procedimiento implica la conversión de datos en formato JSON a instancias concretas de las clases correspondientes a mapaDto y caminoDto, permitiendo así la representación y manipulación de la información contenida en dicho archivo de manera estructurada.

3.7. MVC

3.7.1. MapaVista

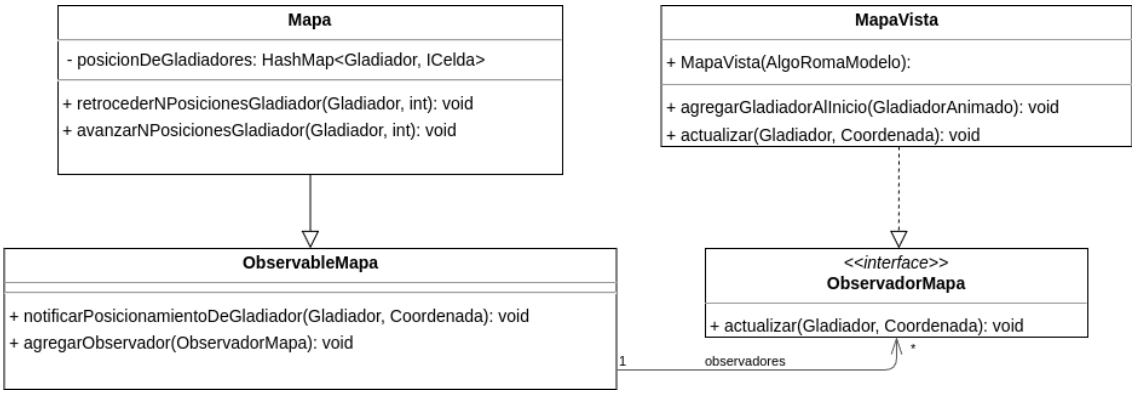

Figura 14: Diagrama de clase: Mapa MVC.

3.7.2. EnergiaVista

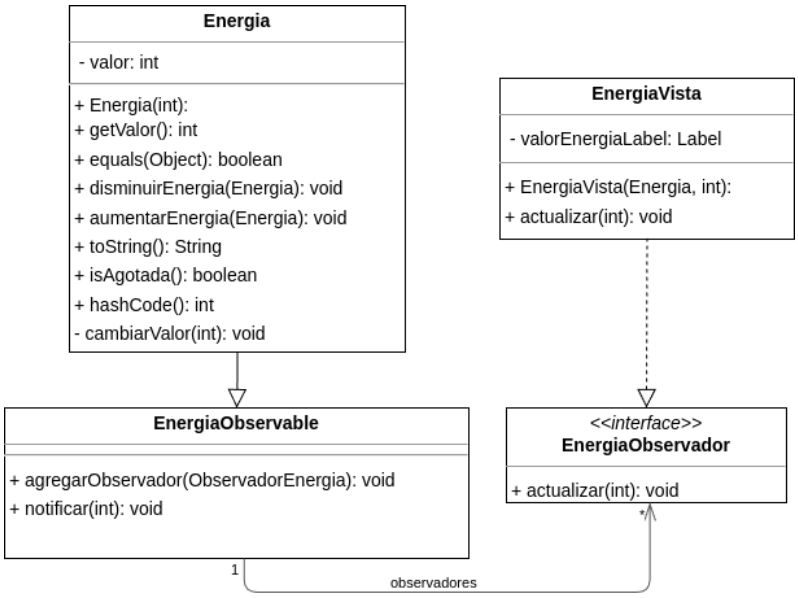

Figura 15: Diagrama de clase: Energia MVC.

3.7.3. SenorityVista

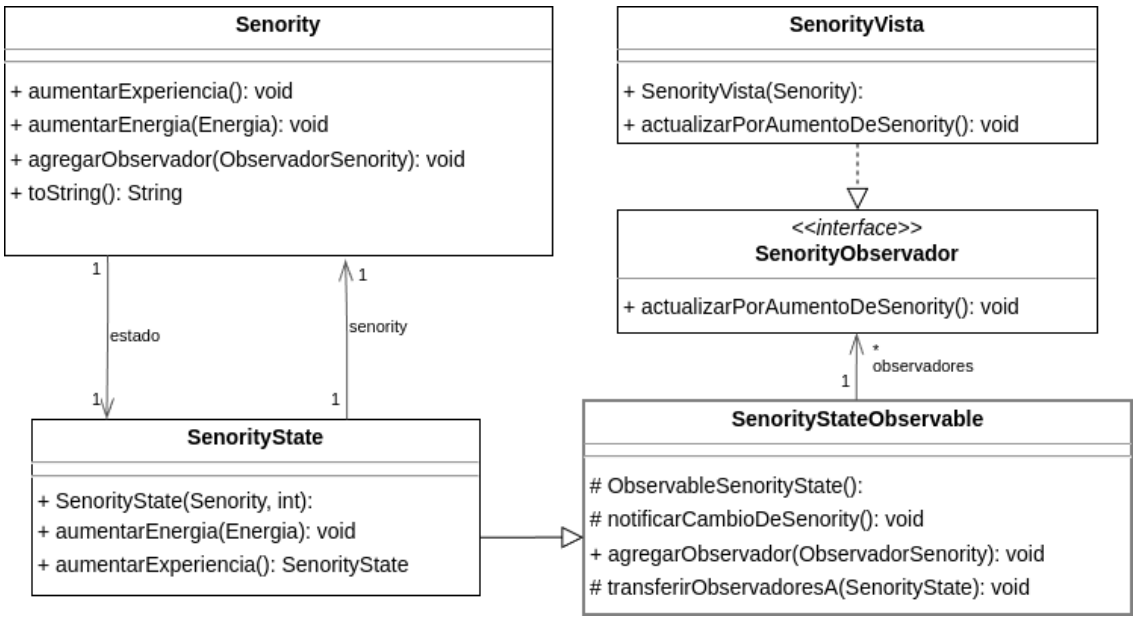

Figura 16: Diagrama de clase: Senority MVC.

3.7.4. Elegir Gladiador

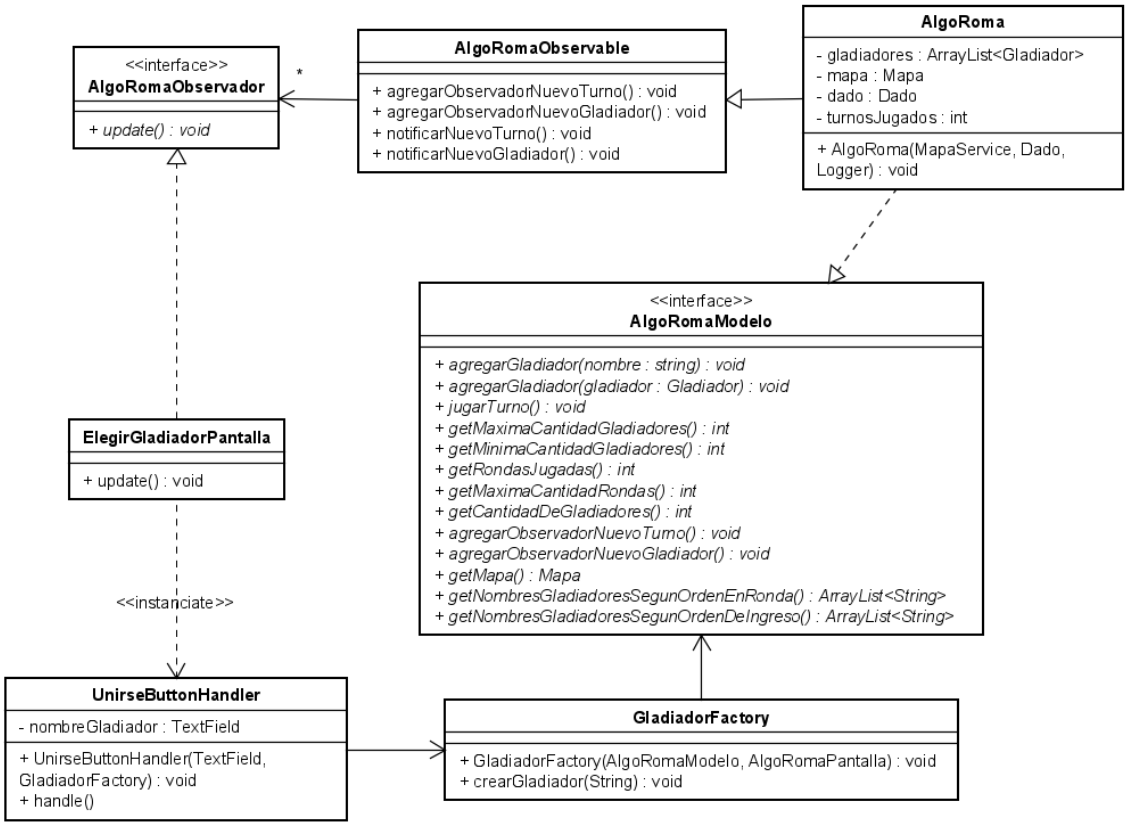

Figura 17: Diagrama de clase: Clases involucradas al elegir gladiador haciendo uso del patron MVC.

3.7.5. DadoButton

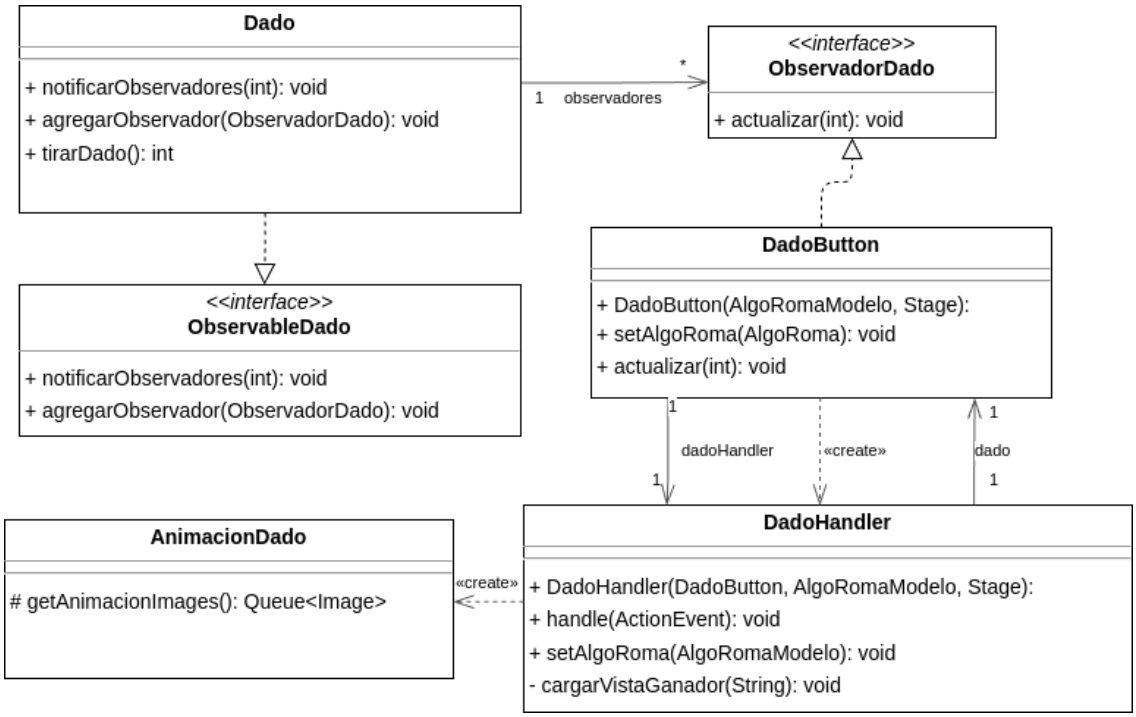

Figura 18: Diagrama de clase: Clases involucradas al usar el dado haciendo uso del patron MVC.

4. Diagramas de secuencia

4.1. Cargar Mapa utilizando MapaService

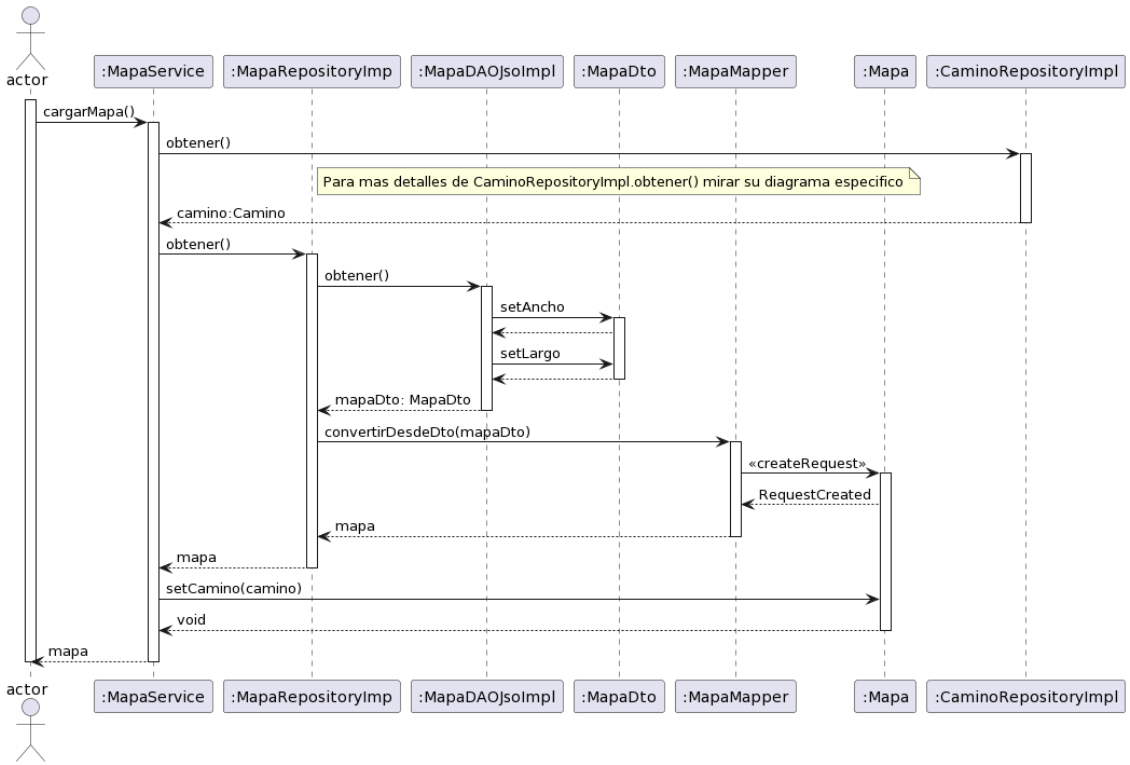

Figura 19: Diagrama de secuencia: Cargar Mapa utilizando MapaService

4.1.1. Cargar Camino con CaminoRepository

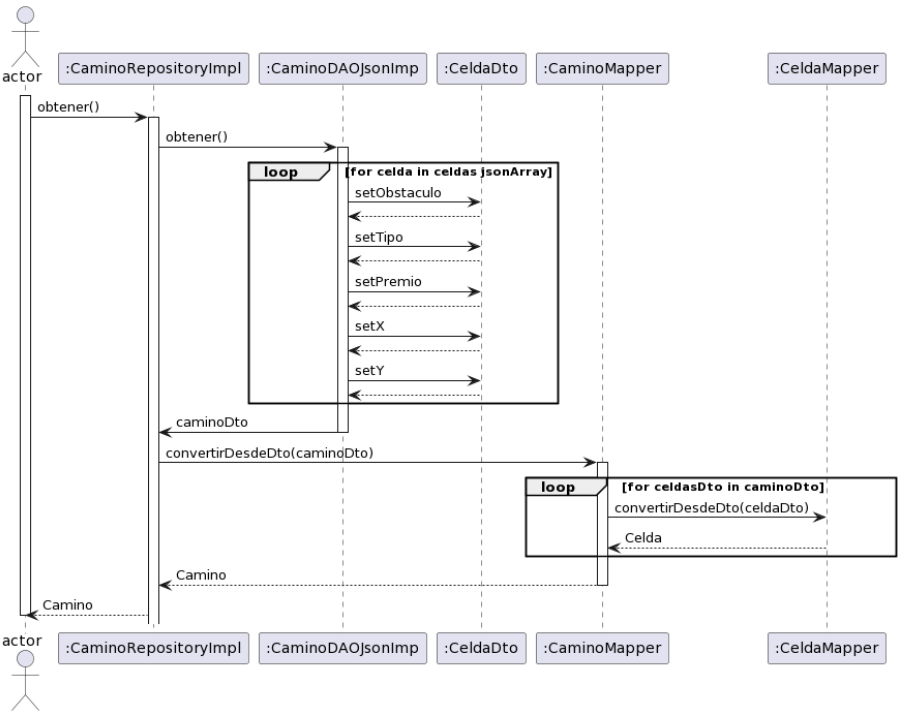

Figura 20: Diagrama de secuencia: Cargar Camino utilizando CaminoRepository.

4.2. Jugar turno

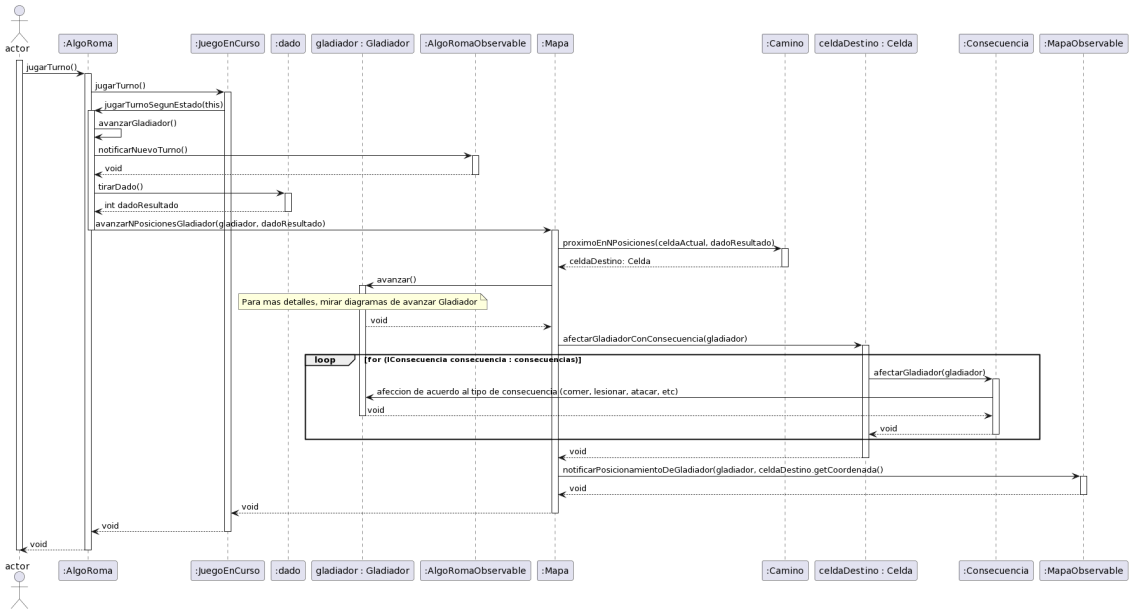

Figura 21: Diagrama de secuencia: Jugar Turno sin detalles de como avanza y se afectan las consecuencias al gladiador

4.3. Consecuencias

El siguiente diagrama de secuencia no busca ilustrar un caso de uso especifico sino mostrar el funcionamiento general de las Consecuencias. Todas estas, representadas en el diagrama de clases 4, implementan el método "afectarGladiador(Gladiador gladiador)" y componen a una Celda, como se muestra en el diagrama de clases 3. De este modo es posible hacer uso del método "afectarGladiadorConConsecuencia(Gladiador:gladiador)" y lograr que el mismo afecte al gladiador con cada Consecuencia presente en la Celda respondiendo a una interface en común, el método "afectarGladiador(Gladiador)", logrando así un comportamiento polimórfico frente a cada de una de las consecuencias existentes y futuras implementaciones.

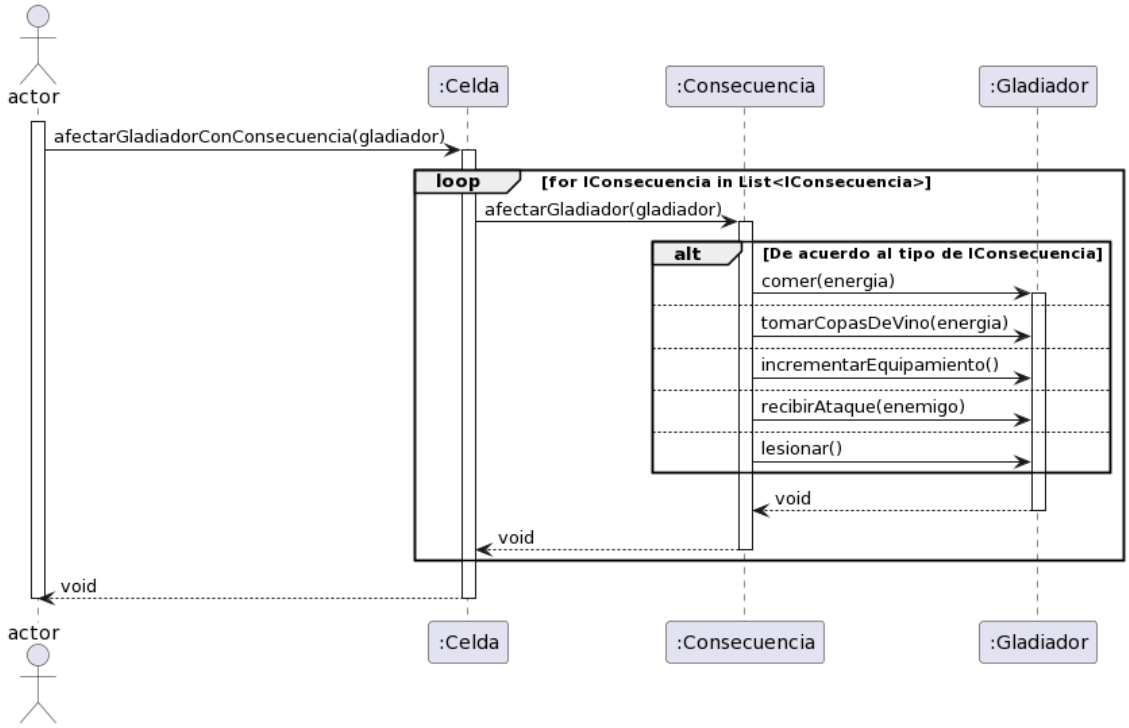

Figura 22: Funcionamiento de las consecuencias.

A continuación se ilustran casos de uso específicos de las consecuencias detallando su comportamiento.

4.3.1. Comida

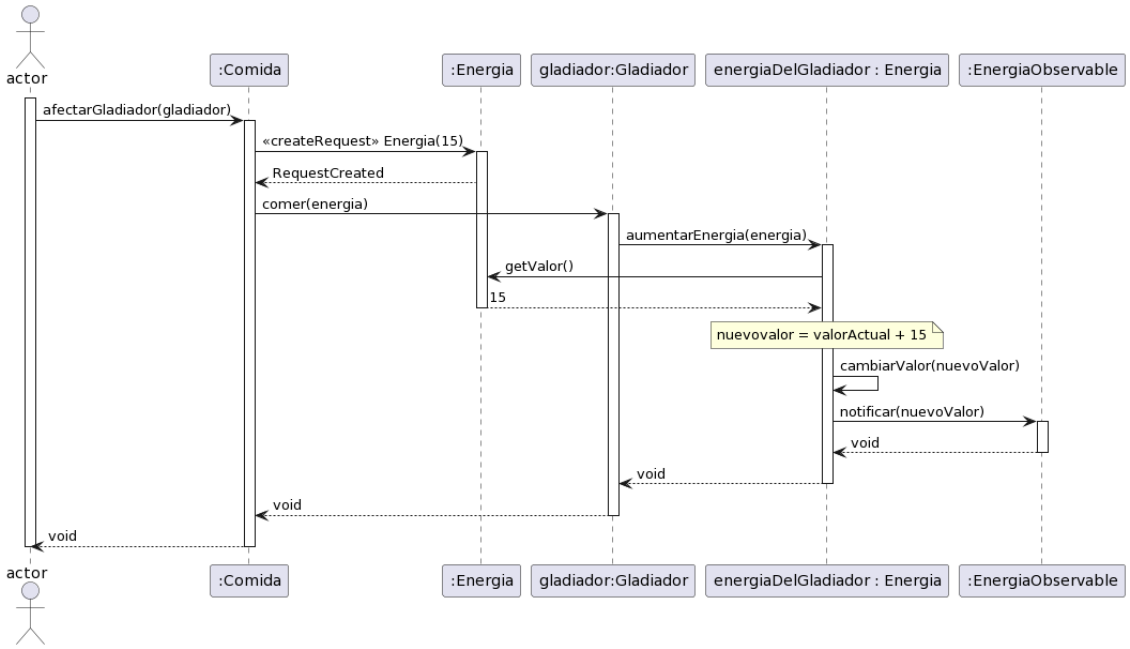

Figura 23: El Gladiador come e incrementa su energia en 15 puntos.

4.3.2. Incremento de equipamiento

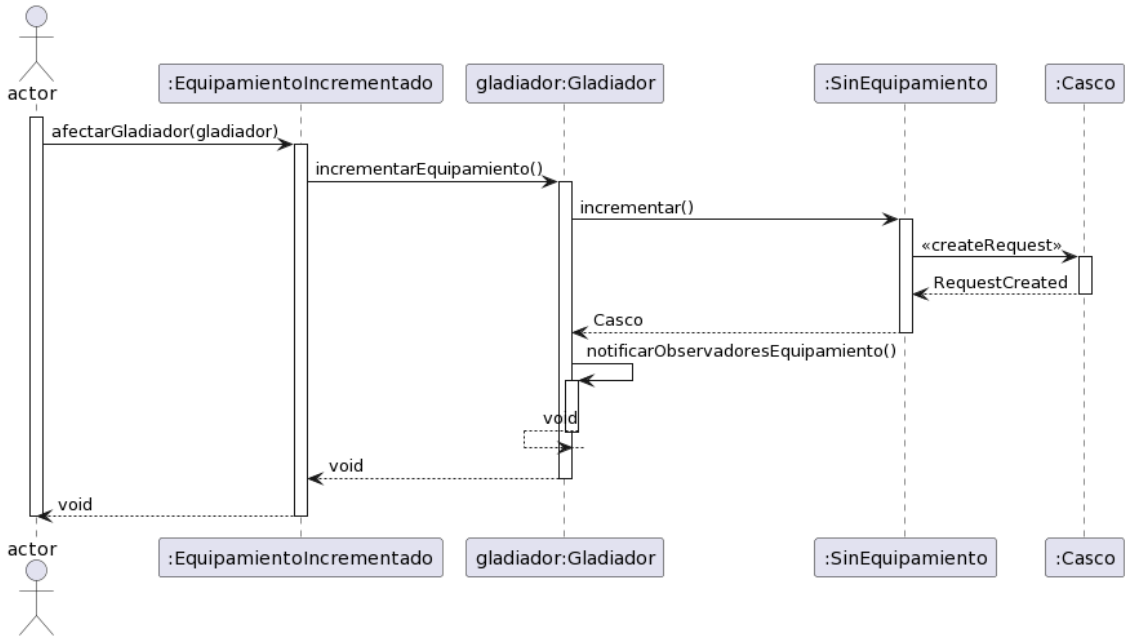

Figura 24: Gladidador sin equipamiento recibe un premio que incrementa el equipamiento, obtiene un casco.

4.4. Ataque de Enemigo

4.4.1. Gladiador es atacado por una fiera salvaje, cuando ya posee todos los equipamientos, no pierde energía

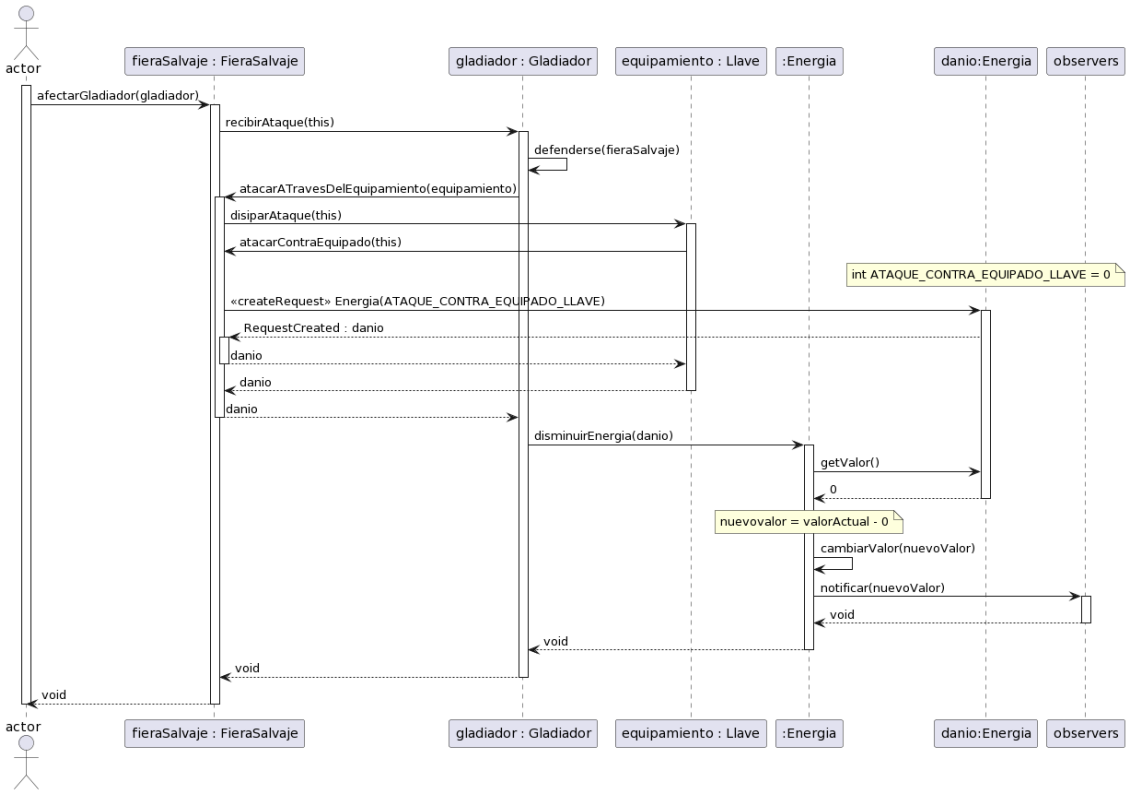

Figura 25: Gladiador es atacado por una fiera salvaje, cuando ya posee todos los equipamientos, no pierde energía.

4.4.2. Gladiador es atacado por una fiera salvaje, cuando posee un casco, pierde 15 puntos de energía

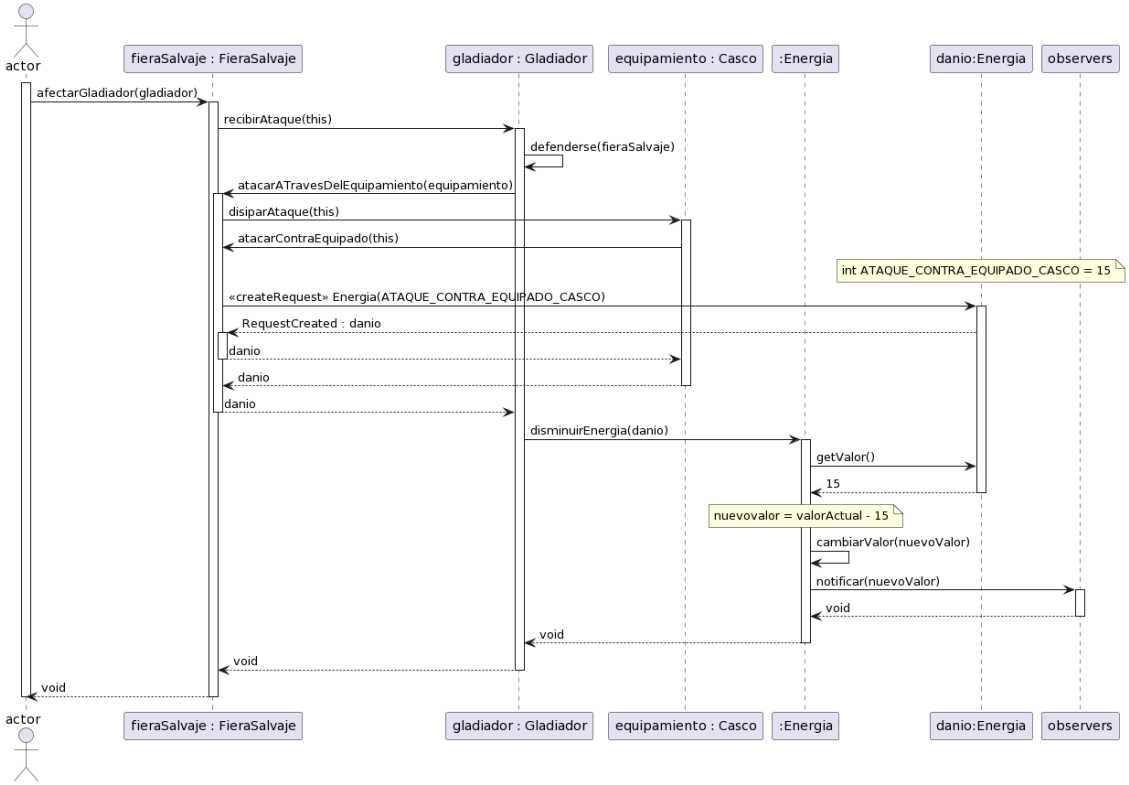

Figura 26: Gladiador es atacado por una fiera salvaje, cuando posee un casco, pierde 15 puntos de energía.

4.5. Senorities

4.5.1. Avance de Gladiador con senority Novato.

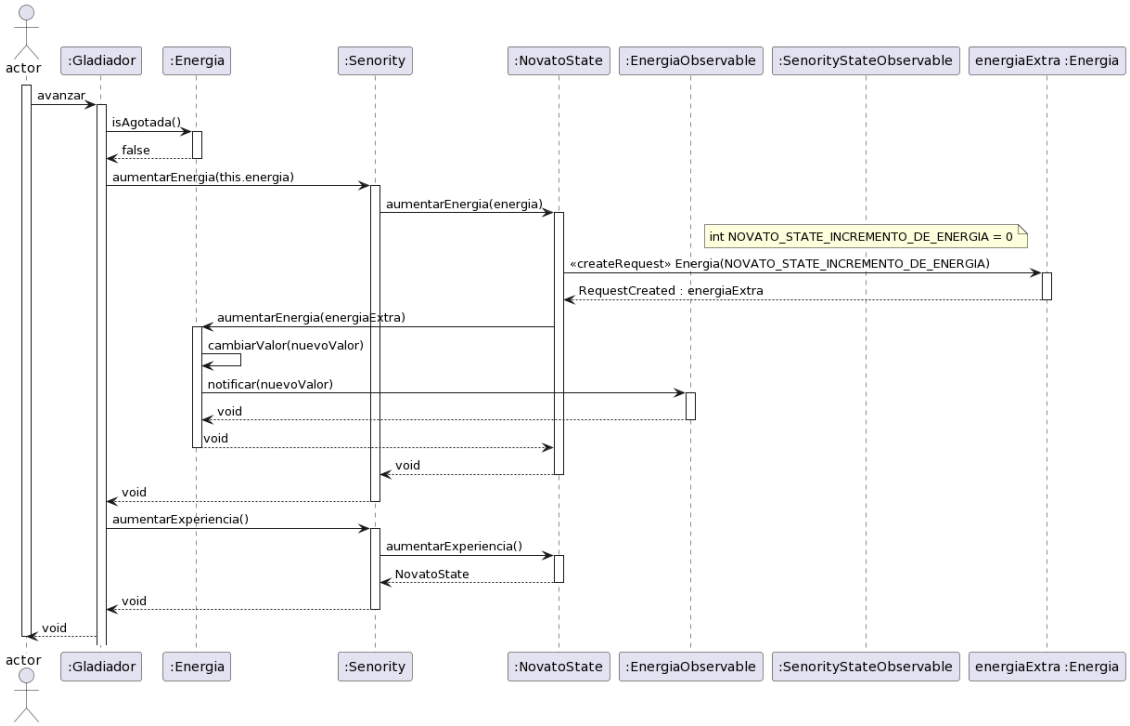

Figura 27: Avance de Gladiador con senority Novato.

4.5.2. Avance de Gladiador con senority Novato que incrementa a SemiSenior

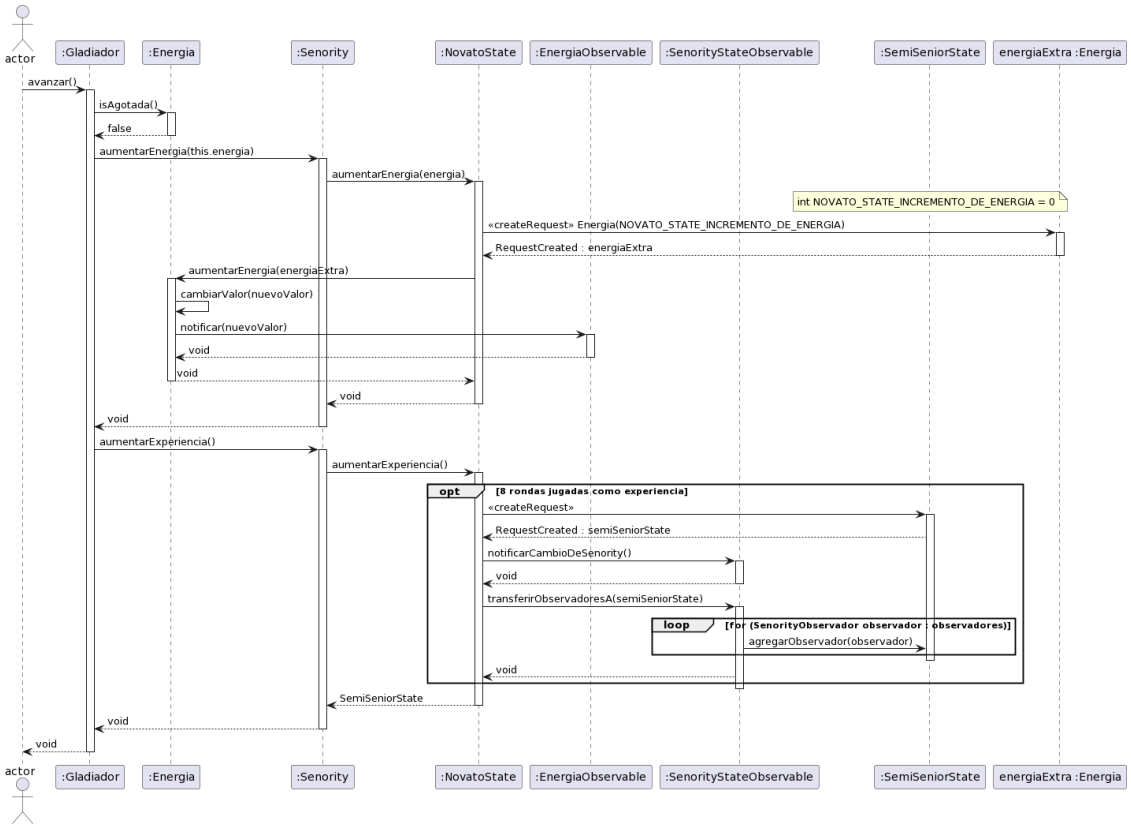

Figura 28: Avance de Gladiador con senority Novato que incrementa a SemiSenior.

4.5.3. Avance de Gladiador con SemiSenior

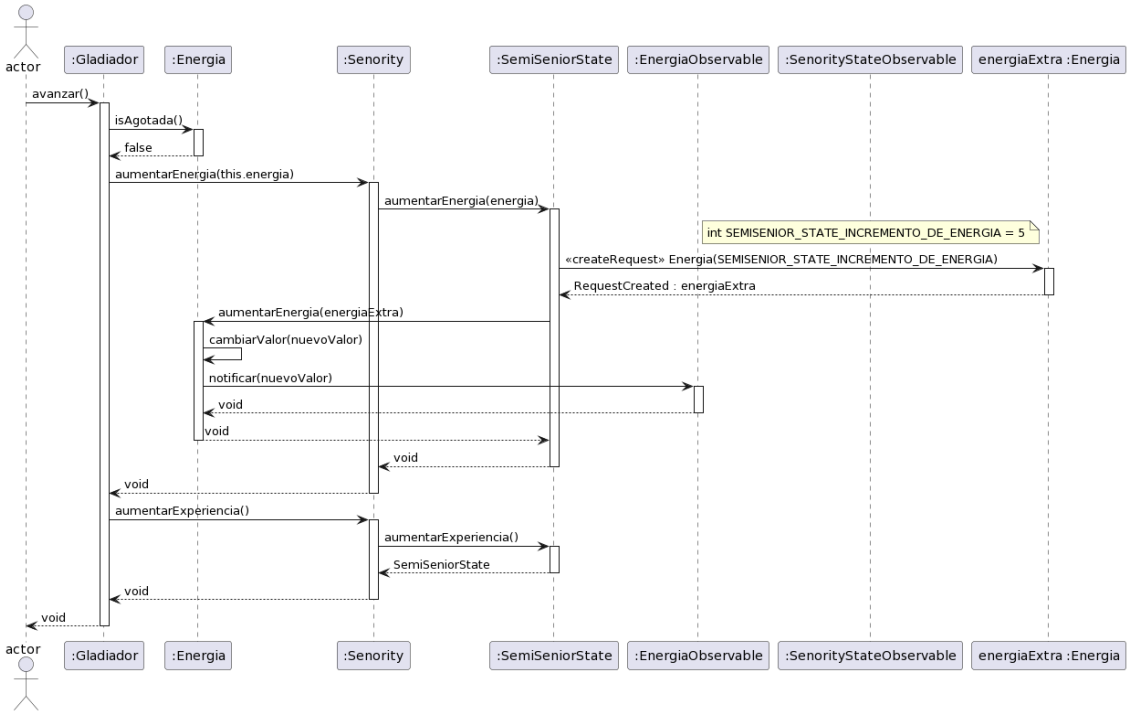

Figura 29: Avance de Gladiador con SemiSenior.

4.6. Pulsar Boton Unirse

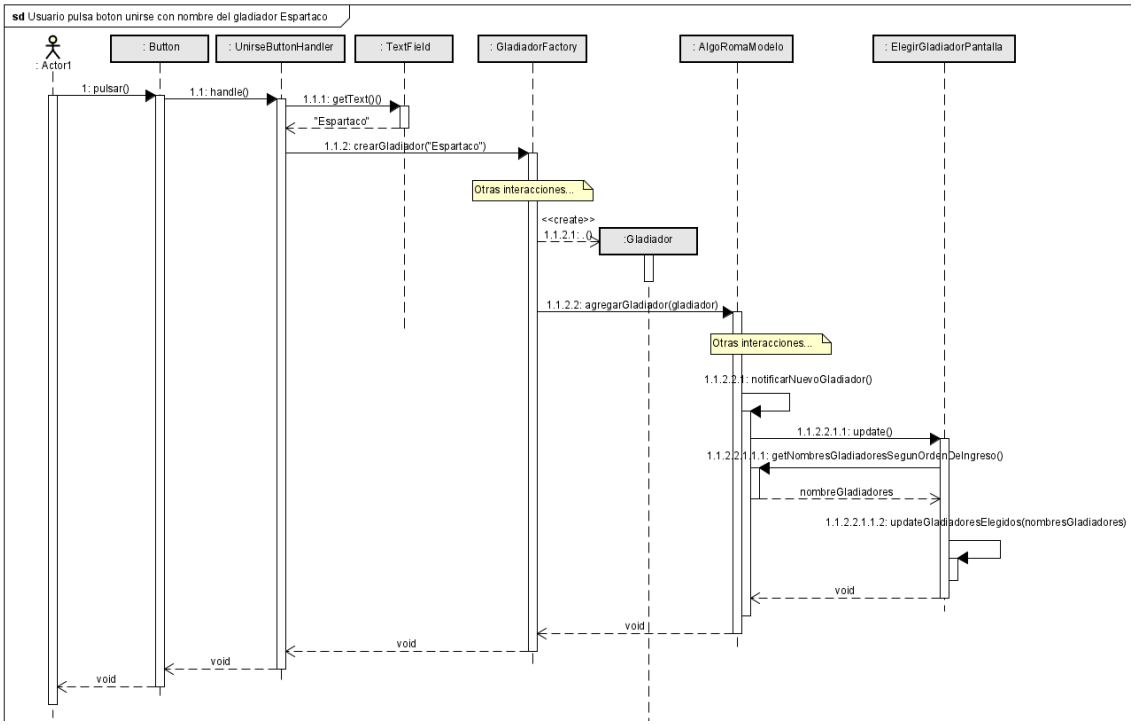

Figura 30: Pulsar Botón Unirse.

4.7. Pulsar Dado

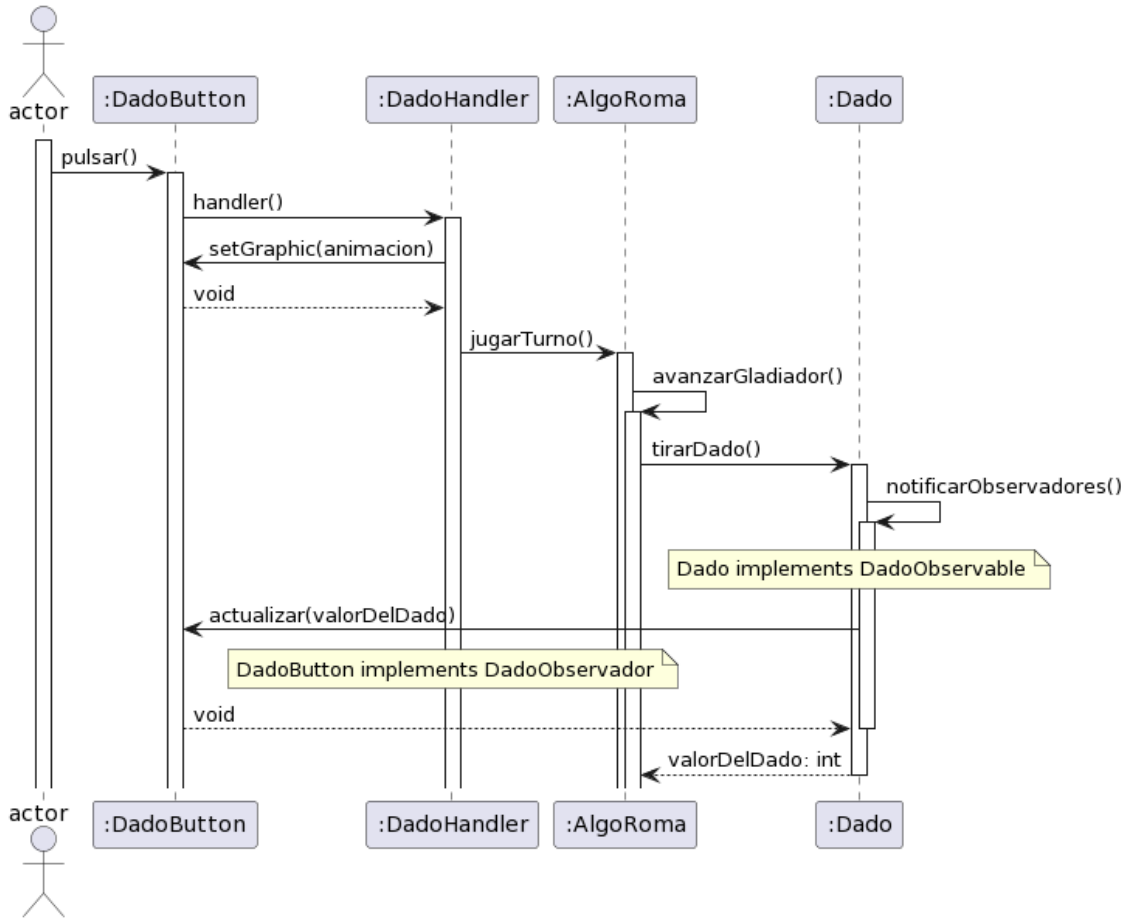

Figura 31: Pulsar Dado.

5. Diagrama de paquetes

En esta sección se presentan los diagramas que permiten visualizar la organización de los distintos paquetes que conforman el proyecto y las relaciones que presentan entre sí.

5.1. Diagrama de paquete: General

En el diagrama de paquetes general (Fig. Nro.:32) se observa la estructura principal orientada a Modelo-Vista-Controlador, la misma facilita la modularidad, el mantenimiento y la escalabilidad. El paquete "resources" no esta asociado a ningun otro paquete debido a que el mismo contiene archivos multimedia (imágenes, animaciones, sonido, etc) y estos son utilizados invocando su ruta absoluta en formato String.

El Modelo no tiene conocimiento directo de la interfaz de usuario ni de como se presentan los datos. La Vista recibe los datos del modelo y muestra los datos en la interfaz de usuario, esto hace que ante cambios en la interfaz de usuario no afecten directamente la lógica central del proyecto. El Controlador es quien actualiza el Modelo según la acción del usuario.

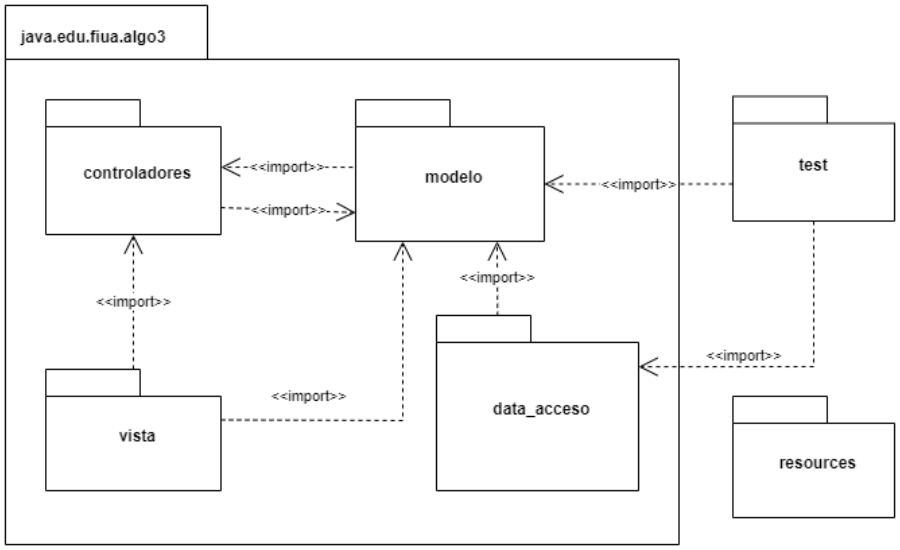

Figura 32: Diagrama de Paquete: General.

5.2. Diagrama de paquete: Modelo

El Modelo representa la estructura de datos y la lógica del proyecto. En la Fig Nro.: 33 se muestran los subdirectorios de "modelo" que se relacionan principalmente con el paquete "gladiador". Si bien en los diagramas se muestran diagramas de clases, los mismos no son mostrados en detalle ya que fueron agregados para una mejor visualización de la relación entre los paquetes.

La Figura Nro.: 34 muestra los paquetes principales que se relacionan con la clase Mapa, incluida en "modelo". Con amabas figuras mencionadas anteriormente se logra ver claramente la organización del paquete "modelo".

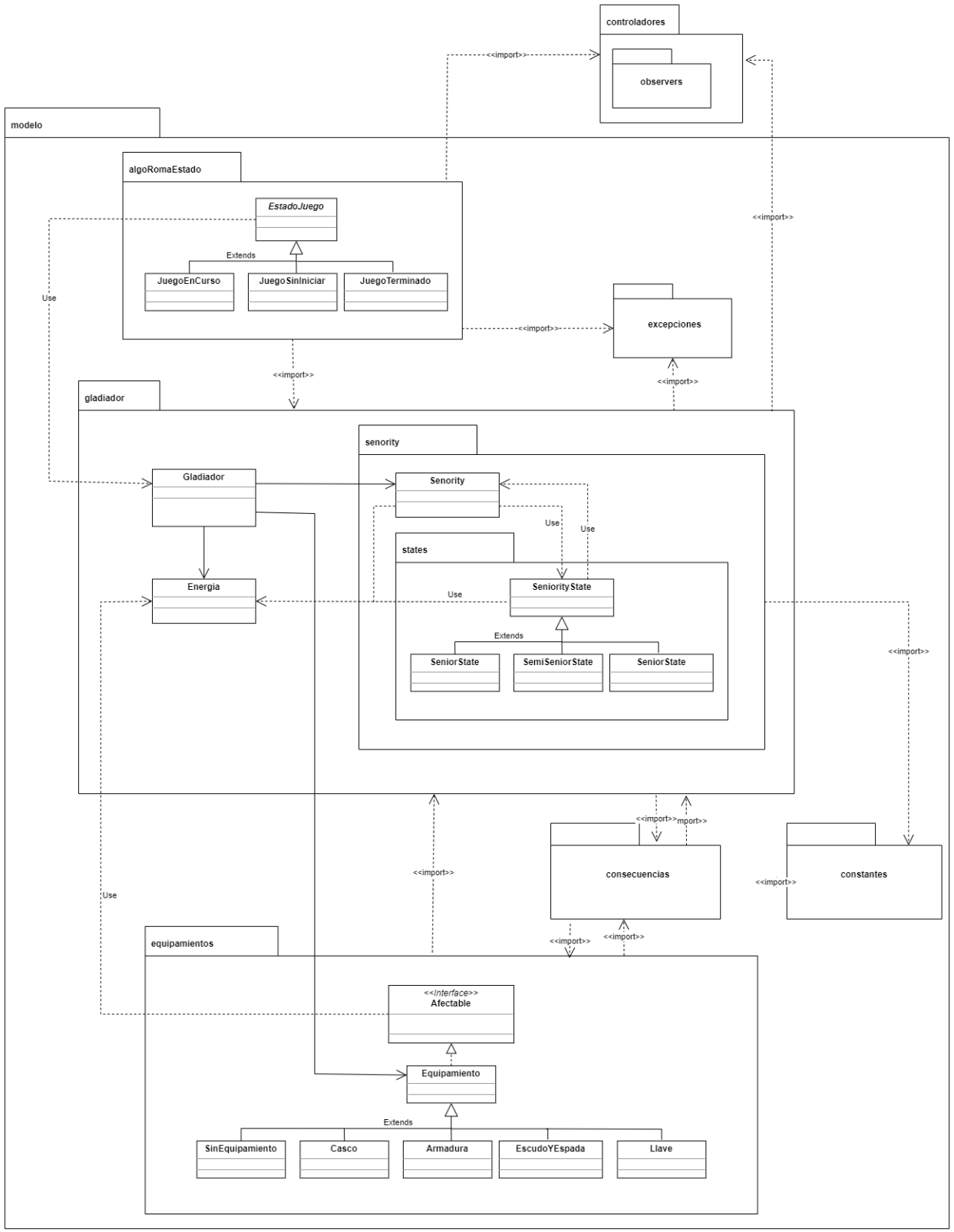

Figura 33: Diagrama de Paquete: Modelo - 1era Parte.

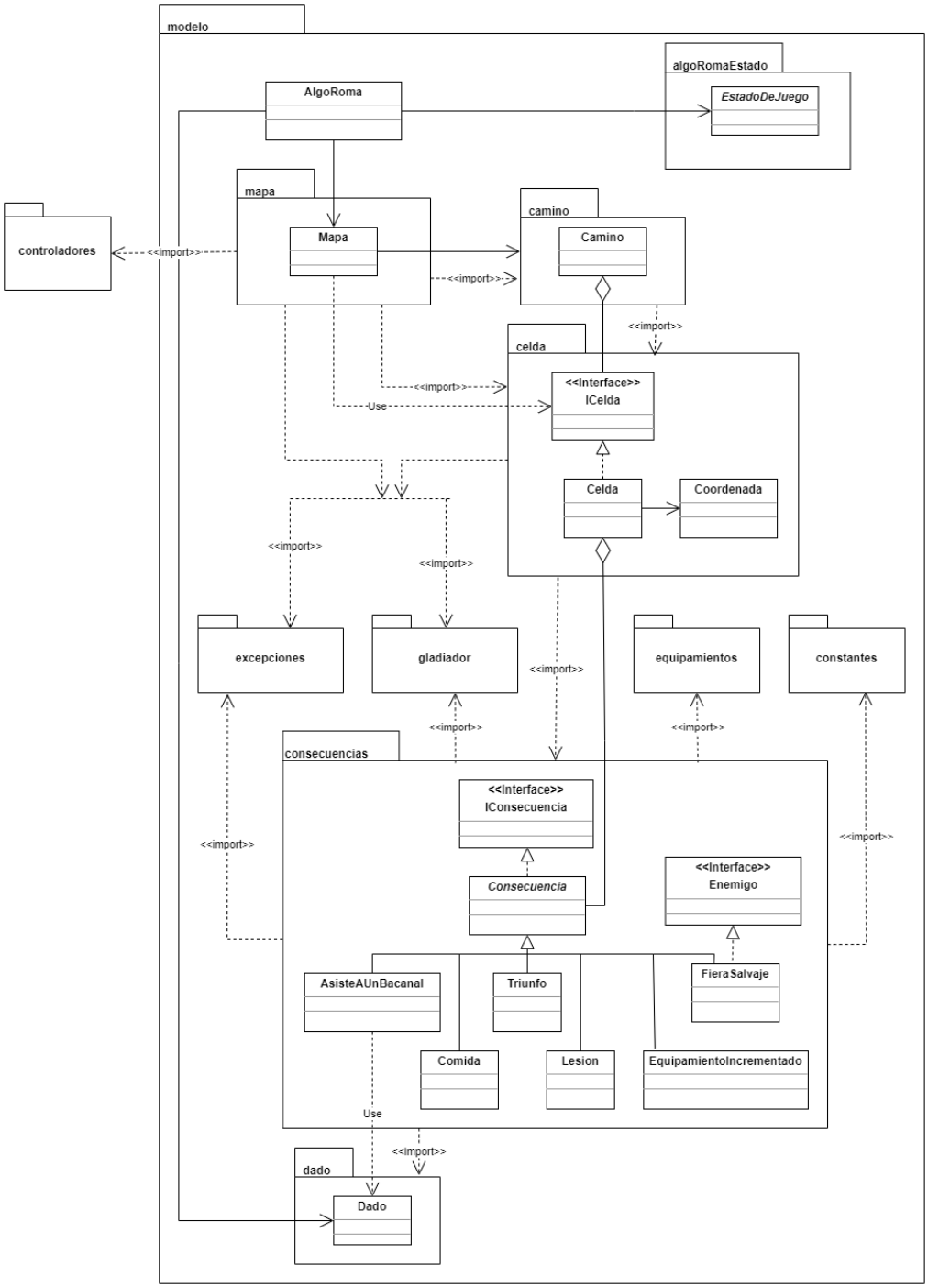

Figura 34: Diagrama de Paquete: Modelo - 2da parte

5.3. Diagrama de paquete: Vista

La Vista es responsable de la presentación y visualización de los datos de usuario. Al recibir información del Modelo, es la encargada de mostrar los datos en la interfaz del usuario. Cuando el usuario interactúa con la interfaz, envía eventos al Controlador pero no realiza operaciones directas sobre los datos del Modelo. El controlador es quien actúa como intermediario entre el Modelo y la Vista, recibe eventos de entrada por parte del usuario desde la Vista y actualiza el Modelo.

En la Fig. Nro.: 35 se muestra como la vista MapaVista se relaciona con el Modelo y los controladores para poder mostrar el estado del mismo al usuario.

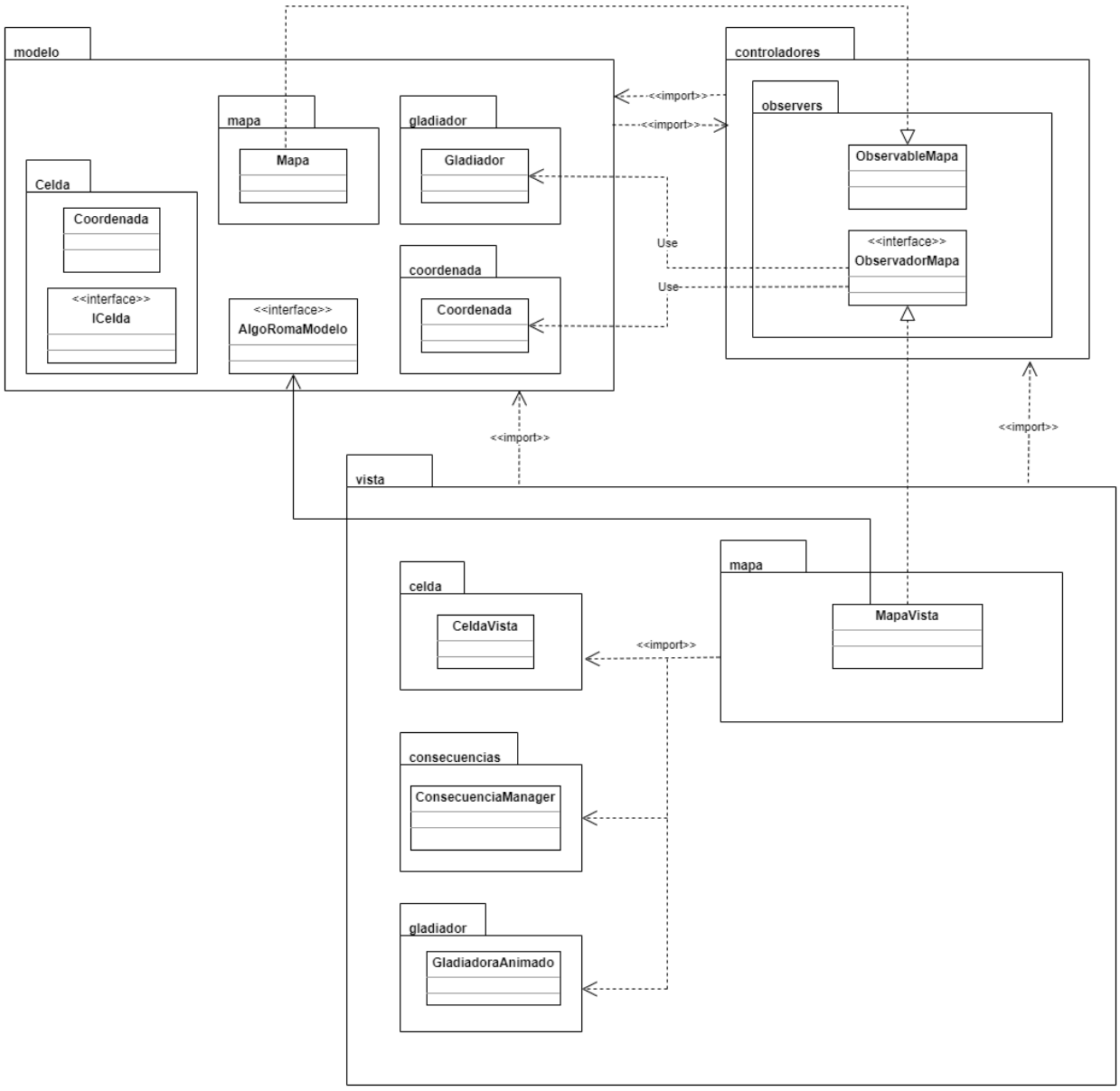

Figura 35: Diagrama de Paquete: MapaVista

5.3.1. Diagrama de paquete: data_acceso

En la Fig Nro.: 36 se muestra la organizacion del paquete "data_acceso" el cual es utilizado para obtener la estructura de datos que conforma los objetos mapaDto y caminoDto que se menciono en la Seccion.: 3.6.2.

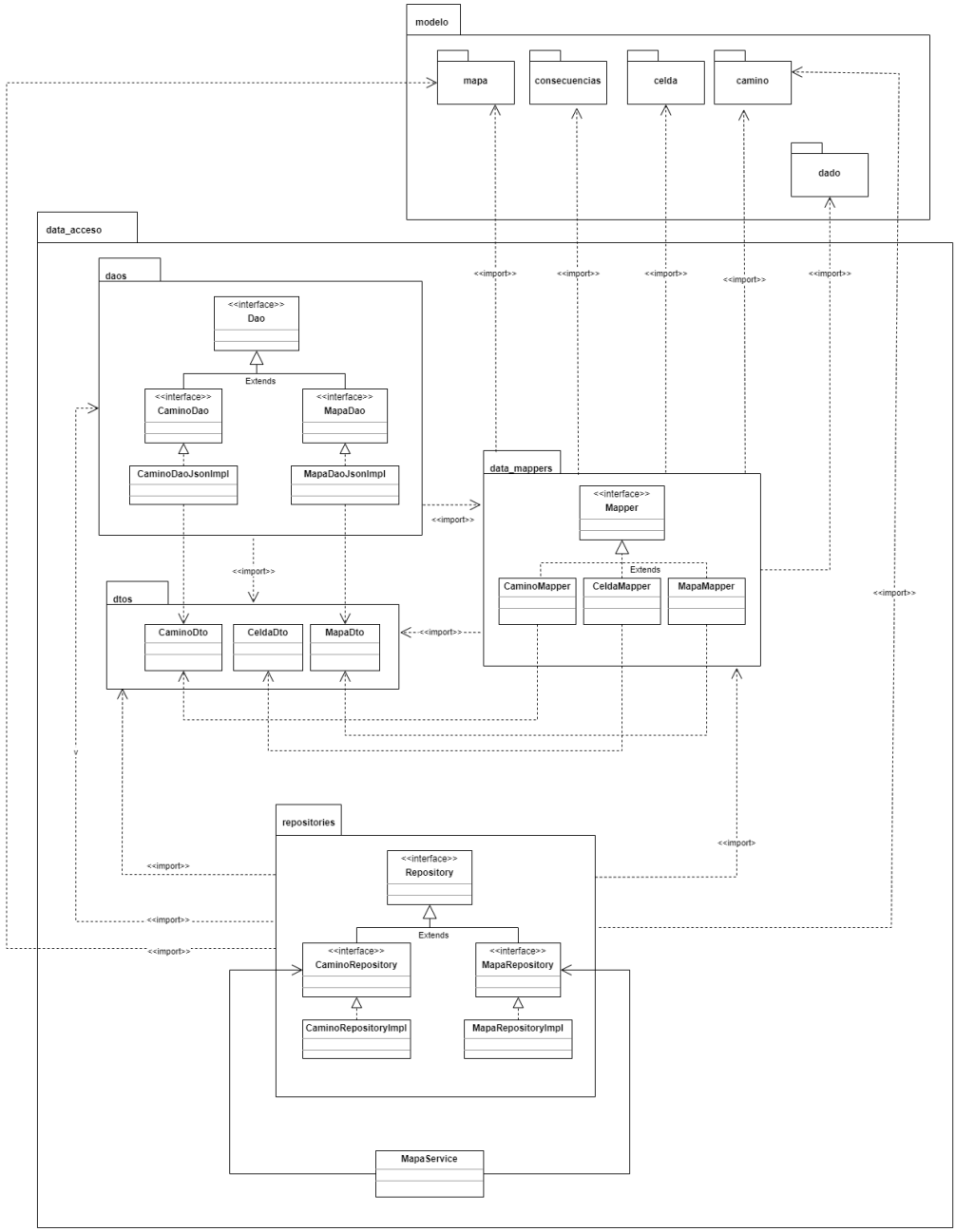

Figura 36: Diagrama de Paquete: data_acceso

6. Diagramas de estado

A continuación se muestran los estados que adoptan algunas entidades u objetos implementadas, en el transcurso del tiempo.

6.1. Diagrama de estados: Senority

Los gladiadores tienen un Senority como se muestra en su diagrama de clase (Sección.: 3.5). El mismo se incrementa por cantidad de turnos jugados. En la Fig. Nro.: 37 se muestra el diagrama de estados.

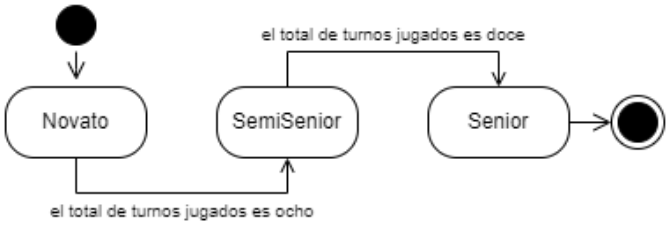

Figura 37: Diagrama de Estados: Senority

6.2. Diagrama de estados: Equipamiento

La Fig. Nro.: 38 muestra los distintos estados perteneciente a Equipamiento. Considerando lo explicado en la Sección.: 3.5.1 el diagrama de estados obtenido es el siguiente:

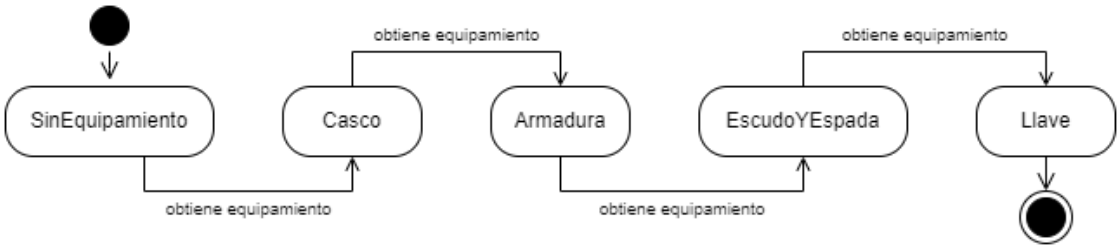

Figura 38: Diagrama de Estados: Equipamiento

6.3. Diagrama de estados: AlgoRomaEstado

La clase AlgoRoma tiene asociada AlgoRomaEstado cuyo estado varia dependiendo si el juego se encuentra inicializado, se haya alcanzado el máximo numero de rondas o unos de los gladiadores ganó el juego. La Fig. Nro.: 39 muestra los estados que adopta en su tiempo de vida.

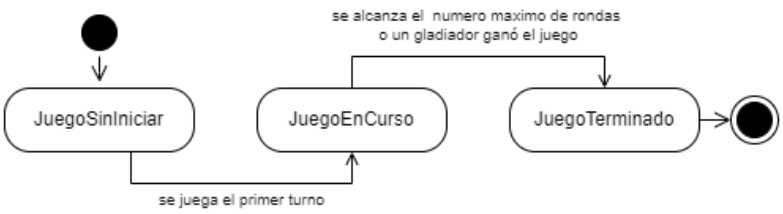

Figura 39: Diagrama de Estados: AlgoRomaEstado

7. Detalles de implementación

7.1. El Gladiador frente a distinto tipos de Enemigos

Ante un requerimiento de este tipo, resulta valioso lograr un modelado que torne sencilla la tarea de implementar un nuevo tipo de enemigo, de modo que decidimos crear la clase abstracta Enemigo, de la cual extiende FieraSalvaje, y darle así una interfaz en común.

Obstáculo	**Energía perdida según equipamiento:**
Fiera salvaje hambrienta	Nada= -20 Casco = -15 Casco, Armadura = -10 Casco, Armadura, Escudo y espada = -2 Llave de la casa = 0

Cuadro 1: comportamiento de fiera salvaje

En conjunto el Gladiador se abstrae del tipo especifico de Enemigo que lo ataca logrando así un comportamiento polimórfico sin necesidad de sobrecargar el método 'recibirAtaque()' (evitamos implementar uno por cada tipo de Enemigo) y desacoplando la clase Gladiador de los Enemigos (Si requerimos un nuevo enemigo, no resulta necesario modificar la clase Gladiador implementando por ejemplo un método 'recibirAtaque(NuevoEnemigo)'). El diagrama de secuencia 26 resulta útil para comprender este comportamiento.

```
public void recibirAtaque(Enemigo enemigo) {
    Energia danioTrasDefenderse = this.defenderse(enemigo);
    this.energia.disminuirEnergia(danioTrasDefenderse);
}

private Energia defenderse(Enemigo enemigo) {
    return enemigo.atacarATravesDelEquipamiento(this.equipamiento);
}
```

De este modo frente a un requerimiento como el siguiente:

Obstáculo	**Energía perdida según equipamiento:**
Leon	Nada= -100 Casco = -80 Casco, Armadura = -60 Casco, Armadura, Escudo y espada = -40 Llave de la casa = 20

Tan solo resulta necesario implementar la clase Leon que extienda de Enemigo y una interfaz que contenga los valores correspondientes frente a cada tipo de equipamiento. Como se muestra a continuación:

```
public interface LeonConstantes {
    int ATAQUE_CONTRA_EQUIPADO_SIN_EQUIPAMIENTO = 100;
    int ATAQUE_CONTRA_EQUIPADO_CASCO = 80;
    int ATAQUE_CONTRA_EQUIPADO_ARMADURA = 50;
    int ATAQUE_CONTRA_EQUIPADO_ESCUDO_Y_ESPADA = 25;
    int ATAQUE_CONTRA_EQUIPADO_LLAVE = 10;
}
```

```
public class Leon extends Enemigo implements LeonConstantes {

    @Override
    public void afectarGladiador(Gladiador gladiador) {
        gladiador.recibirAtaque(this);
    }

    @Override
    public Energia atacarATravesDelEquipamiento(Equipamiento equipamiento) {
        return equipamiento.disiparAtaque(this);
    }

    @Override
    public Energia atacarContraEquipado(SinEquipamiento sinEquipamiento) {
        return new Energia(ATAQUE_CONTRA_EQUIPADO_SIN_EQUIPAMIENTO);
    }

    @Override
    public Energia atacarContraEquipado(Casco casco) {
        return new Energia(ATAQUE_CONTRA_EQUIPADO_CASCO);
    }

    @Override
    public Energia atacarContraEquipado(Armadura armadura) {
        return new Energia(ATAQUE_CONTRA_EQUIPADO_ARMADURA);
    }

    @Override
    public Energia atacarContraEquipado(EscudoYEspada escudoYEspada) {
        return new Energia(ATAQUE_CONTRA_EQUIPADO_ESCUDO_Y_ESPADA);
    }

    @Override
    public Energia atacarContraEquipado(Llave llave) {
        return new Energia(ATAQUE_CONTRA_EQUIPADO_LLAVE);
    }
}
```

Consideremos como responde equipamiento al método 'disiparAtaque()', en el caso en donde equipamiento es un EscudoYArmadura.

```
public class EscudoYEspada extends Equipamiento {
    @Override
    public Energia disiparAtaque(Enemigo enemigo) {
        return enemigo.atacarContraEquipado(this);
    }
}
```

Este modelado no escapa de la necesidad de sobrecargar el método 'atacarContraEquipado()' en función de cada tipo de equipamiento existente, provocando que cada vez que se añada un equipamiento nuevo se deba añadir al enemigo una nueva sobrecargar para método 'atacarContra-Equipado(nuevoEquipo)'. Sin embargo no encontramos posible escapar de esta situación ya que de acuerdo al enunciado los equipamiento no posee un comportamiento especifico (por ejemplo: Casco reduce a la mitad el daño) de modo tal de poder delegar el comportamiento en el mismo, sino que simplemente cada enemigo causa un daño diferente frente a los distintos equipos (como

se muestra en el cuadro 1) quedando entonces la responsabilidad de actuar polimorficamente en el Enemigo de acuerdo al tipo de equipamiento.

Es importante mencionar que la razón por la cual buscamos evitar la sobrecarga de métodos para responder polimorficamente es que en situaciones como lo es agregar un nuevo tipo de Equipamiento viola el principio abierto-cerrado, es necesario modificar todos los enemigos para agregar un nuevo método que resuelva la colisión. Esto es evitado satisfactoriamente en el caso del Gladiador frente a la situación en donde se implementa un nuevo Enemigo.

8. Excepciones

TriunfoNoPosibleException El Gladiador no puede ganar la partida porque no posee el equipamiento requerido para hacerlo. Es lanzado por el método triunfar() de la clase Gladiador, al se llamada por la Consecuencia 'Triunfo' presenta en la ultima celda del camino, si el Gladiador no posee como equipamiento al de mas alta jerarquia (llave).

FinDelJuegoException El juego ha finalizado. Es lanzada al superar el maximo de turnos permitidos.

NombreInvalidoException El nombre del gladiador no es valido. Es lanzada al instanciar un Gladiador con un nombre menor a 4 caracteres o con un nombre previamente registrado en otro gladiador.

MovimientoException El movimiento no puede ser realizado. Es lanzado por el metodo avanzar del Gladiador cuando el mismo no puede moverse. Puede ser por falta de energía o por que se encuentra lesionado (el mensaje del mismo lo detalla).

JsonFormatoInvalidoException Es utilizado por el paquete "data accesoçuando el Json no tiene el formato esperado.

JuegoEnCursoException El juego esta en curso. Es lanzado por AlgoRoma cuando se buscar añadir un nuevo gladiador a un juego ya iniciado.

MaximoGladiadoresException Es lanzado por AlgoRoma cuando se buscar añadir un gladiador excediendo así el numero máximo de Gladiador permitidos para jugar.

MinimoGladiadoresException Es Lanzado por AlgoRoma cunado se busca iniciar el juego sin el numero suficiente de Gladiadores para jugar.