UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de un videojuego casual mediante un sistema multi-agente SPADE.

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Vidal Ramón, Pau

Tutor/a: Carrascosa Casamayor, Carlos

CURSO ACADÉMICO: 2022/2023

Creación de un videojuego casual mediante un sistema multi-agente SPADE

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Pau Vidal Ramón

Tutor: Carlos Carrascosa Casamayor

2022/2023

Resumen

Este proyecto trata sobre la unión de diferentes tecnologías para la creación de un sistema de combate de un videojuego casual de cartas. En este, se verá la gestión de una ontología OWL para la creación del sistema de mazos y cartas. Además, como punto central del proyecto se encuentran los agentes inteligentes, programados con SPADE, los cuales representan estas cartas. Para poder visualizar la amplitud de este proyecto e interactuar con los agentes y ontología se ha usado el motor gráfico de Unity, vinculado a las anteriores tecnologías mediante el uso de sockets TCP.

Palabras clave: videojuego casual, Unity, agentes inteligentes, SPADE, ontología OWL, socket TCP.

Abstract

This project involves the integration of different technologies to create a combat system of a casual card video game. The project focuses on the management of an OWL ontology for the creation of the decks and cards system. In addition, as the central point of the project are the intelligent agents, programmed with SPADE, which represent these cards. To visualize the amplitude of this project, and to interact with the agents and the ontology, the Unity game engine has been employed, linked to the previous technologies using TCP sockets.

Keywords: casual videogame, Unity, intelligent agents, SPADE, OWL ontology, TCP socket.

Tabla de contenidos

Lista de figuras

1. Introducción

1.1. Motivación

El principal motivador de este proyecto fue la idea de crear y programar un videojuego desde cero, que hiciese uso de algún elemento relacionado con la inteligencia artificial. Para esto se decidió realizar un videojuego casual de estrategia de batallas de cartas por turnos. Este trabajo está centrado en el sistema de combate del videojuego, siendo las cartas agentes inteligentes y usando un motor gráfico para representar el combate.

Con esto en mente, se adhirió la complejidad de poder vincular varias tecnologías aparentemente inconexas entre sí y hacer que trabajen juntas. El desafío de conseguir hacer un proyecto que englobe estas tecnologías y hacer que trabajen juntas sirvió también como una fuerte motivación, ya que se vio como un reto a superar.

1.2. Justificación

Se escogió programar el sistema de combate un juego de batallas de cartas con agentes por varias razones. Entre ellas, la popularidad de los juegos de cartas de esta índole a nivel mundial, por lo tanto, en previsión de sacarlo a mercado podría entrar como un videojuego original en su idea y, por tanto, competente. Otra razón es la posibilidad de ampliación a futuro con nuevos modos de juego usando este mismo sistema de combate, permitiendo así un mayor nivel de flexibilidad y polivalencia, tanto entre los jugadores como entre la competencia del mercado, ya que abarcaría un rango mayor entre diferentes tipos de videojuegos. [16]

Hay que comentar, antes que nada, y a raíz del punto anterior, que este proyecto no se comercializará, pero sus tecnologías pueden perfectamente integrarse a otros proyectos, sobre todo estando pensado en integrarse al proyecto de Enric Puigcerver, con quien se iba a colaborar en un principio para realizar el proyecto.

1.3. Objetivos

El objetivo principal es la creación de un sistema de combate de cartas, dónde los agentes inteligentes actúan de forma autónoma, listo para usar en videojuegos en un entorno gráfico proporcionado por Unity. Así como un entorno que permita gestionar las cartas de los usuarios de este videojuego.

Para lograr este objetivo, de forma paralela se han de cumplir unos objetivos secundarios pero necesarios, tales son:

- Creación de un sistema de sockets para poder cumplir con las comunicaciones por red necesarias para la transmisión de datos entre agentes y entorno gráfico. Esto se entiende

tanto para la gestión del propio usuario para acceder a la ontología como para la comunicación entre agentes.
- Creación de una UI (*User Interface*) dedicada a la interacción de datos entre el motor gráfico de Unity y la ontología y los agentes. Para así poder dotar al usuario de libertad por la aplicación y el poder de elección en ciertos elementos clave del videojuego.
- Creación de una ontología mediante OWL y su respectivo uso y gestión desde la UI. Siendo esto posible mediante el sistema de comunicaciones vía sockets. Esto dota al videojuego de un sistema de gestión de usuarios, colecciones de mazos y cartas y, al mismo tiempo, toda la estructura de atributos de las cartas la cual será la base de los agentes.

1.4. Estructura

La estructura de esta memoria está pensada a explicar en un principio las bases de las tecnologías e ir extendiendo estas mismas con relación a la implementación dentro del proyecto, para finalizar con la relación de estas tecnologías entre ellas. De esta manera, cada apartado aportará algo al siguiente y se podrá seguir con una mayor facilidad el progreso del proyecto.

En primera instancia se introducirá el proyecto, donde se explicará a grandes rasgos de qué trata y qué tecnologías se usan. Se hablará también de las principales motivaciones de este, junto con justificaciones del por qué se ha decidido resolver de esta manera y los objetivos que se buscaban con la realización de este.

Antes de empezar a hablar sobre las tecnologías usadas, se dedicará un apartado a hablar sobre el estado del arte, tanto de sus principales referentes como de su competencia. De esta manera, se entenderá un poco mejor sus raíces y lo que se pretende conseguir.

Seguidamente, en un pequeño apartado se hablará sobre la propiedad intelectual dentro del proyecto, esto se hace para aclarar algunas dudas que puedan surgir y referenciar contenido en el que se ha apoyado el proyecto a la hora de construirlo.

A continuación, se hablará sobre las tecnologías usadas, refiriéndose a estas por sus definiciones y usos en el proyecto, para entender el por qué se ha usado cada una de estas y como se ha adaptado al trabajo final. Cuando se acabe la sección, se conocerán las herramientas que se han usado para la construcción del proyecto y, por tanto, se entenderá mejor el resto del trabajo.

El siguiente apartado tratará sobre diseño de la arquitectura la arquitectura, centrándose en cómo están construidas y relacionadas las diferentes herramientas que componen el proyecto. Para esto se profundizará en los módulos que compone cada parte del proyecto, aportando diagramas UML de clases y figuras. También se hablará de cómo funcionan las comunicaciones de red entre sockets. Para finalizar este apartado se entenderá más en profundidad que tecnologías se han usado para su desarrollo. En este apartado, también se hará hincapié en cómo han de inicializarse y comportarse las distintas herramientas en un caso real, viendo que este importante proceso se realiza de forma implícita.

Entre los próximos apartados se hablará de cómo se ha resuelto la implantación del proyecto, así de qué pruebas se hacían a lo largo de la construcción de este. Finalmente, se

cerrará el trabajo con una conclusión sobre este, los posibles trabajos futuros que se podrían aplicar sobre el proyecto y, para finalizar, los agradecimientos.

1.4. Colaboraciones

Desde un inicio, este proyecto iba a llevarse a cabo en conjunto con Enric Puigcerver Ibáñez. En esta colaboración, este proyecto sería el encargado de crear un sistema de combate mediante agentes inteligentes y la forma de conectarlo con un videojuego creado con Unity, el cual se verá a lo largo del proyecto, y el proyecto de Enric se encargaría de crear el videojuego al cual integrarlo.

Esta colaboración no se pudo llevar a cabo en su momento y, al igual que al proyecto de Enric Puigcerver le faltó por implantar el sistema de combate, a este proyecto le faltaría gran parte del apartado visual, ya que de este se encargaría Enric. Actualmente, en este proyecto se están usando *placeholders* en los recursos gráficos, ya que este solo trata del apartado técnico en referente a agentes inteligentes y conexiones.

2. Estado del arte

En este apartado se va a comentar sobre cómo se decidieron los diferentes sistemas para crear el videojuego, en referente a lo ya existente en el mercado, se compararán las funcionalidades de estos sistemas con las elegidas en el proyecto. Asimismo, se buscará la posible competencia que pueda llegar a haber en un futuro y el cómo y por qué este proyecto podrá sobresalir en un mercado ya situado.

2.1. Análisis del mercado

Para la creación de este videojuego, se han cogido referentes de muchos ámbitos, pasando por una variedad muy amplia de videojuegos e incluso de juegos de mesa. En un principio la idea siempre fue clara, un sistema de combate basado en el famoso juego de rol de mesa ***Dungeons and Dragons***.

Ahora con este sistema en mente, se hubo de pensar en cómo llevarlo a cabo. Como se trata de un sistema basado en turnos se tuvo como primera idea hacer un juego de cartas. Para esto se comparó con los títulos que ya existían en el mercado. Tanto en juegos de mesa como en videojuegos había una gran variedad, entre los más famosos se encuentran:

- **Magic: The Gathering**.

Figura 2.1: Magic Arena. Fuente: geekwire

- **Hearthstone**.

Figura 2.2: Heartstone. Fuente: areajugones

- **Pokémon**.

Figura 2.3: Pokémon. Fuente: wargamer

- **Keyforge**.

Figura 2.4: Keyforge. Fuente: gamingtrend

- **Yu-Gi-Oh!**

Figura 2.5: Yu-Gi-Oh! Duel Links. Fuente: Pocket Gamer

- **Legends of Runaterra.**

Figura 2.6: Legends Of Runaterra. Fuente: millenium.gg

Todos ellos tienen un sistema de combate muy parecido, así como los sistemas de puntuación. En todos ellos, el sistema de combate se trata de una sucesión de turnos, donde cada jugador escoge las cartas que va a jugar en su turno y las deposita en la mesa pagando el coste de la carta (si es que tuviese). Para este proyecto se decidió escoger esta misma fórmula.

Para lo que es el combate, en cada turno de jugador se escogen las cartas que van a atacar o defender, y los hechizos o artefactos que se van a usar. Posteriormente, se ejecutan estas acciones y, progresivamente, se va decantando por un jugador u otro, dependiendo del sistema de puntuación, del que ya se hablará más adelante.

En este momento se llegó a plantear seriamente como este proyecto podría diferenciarse de la competencia, y sobresalir como un juego innovador. Por eso se buscaron otras opciones, y surgió la idea de que el sistema de este videojuego fuese parecido a un auto-battler.

El auto-battler, como su nombre en inglés indica, es un ajedrez automático. Es decir, cada jugador dispone de una cantidad de piezas para colocar en un tablero y estas se pueden mover libremente por este. Al final de cada ronda, el jugador puede decidir si adquirir nuevas piezas o mejorar las que tiene mediante un sistema monetario interno propio del juego. [17]

Entre los juegos más famosos de este género se encuentran:

- **Auto Chess**.

Figura 2.7: Auto Chess. Fuente: Polygon

- **Dota Underlords**.

Figura 2.8: Dota Underlords. Fuente: GameSpot

- **Teamfight Tactics**.

Figura 2.9: TeamFight Tactics. Fuente: Redbull

- **Hearthstone Battlegrounds**.

Figura 2.10: Hearthstone Battlegrounds. Fuente: Reddit

Con las dos opciones en mente, no se llegó a optar finalmente por una, por una parte, el sistema de colecciones propio de un juego de cartas era atractivo. Elementos como la gestión de mazos y cartas son muy llamativas para los jugadores. Por otra parte, el sistema de auto-battler es muy novedoso y se entraría en un mercado por explotar. Con estas dudas en mente, y ante la voluntad de hacer algo nuevo, surgió la idea de mezclar ambos tipos de juego.

Ahora solo quedaría elegir el sistema de puntuación, donde la extensa mayoría de juegos comparten un sistema de puntuación contado por puntos de vida del propio jugador, es decir, el primer jugador que llegase a cero puntos de vida resultaría en el perdedor. Al tratarse este juego de un juego casual y, por tanto, un juego con corta duración se optó por un sistema de puntuación más simple y conocido. Una serie de combates donde la victoria se decantaría al mejor de tres puntos, cada punto ganado al finalizar un combate.

El juego final, sería un juego con un sistema de colecciones típico de los juegos de cartas y con un sistema de combate propio de un *auto-battler*, donde las cartas serían las piezas para jugar y estarían programadas con agentes inteligentes.

Decidido el tema del videojuego, se puede observar que la propia competencia es todos aquellos juegos de los que se ha basado este proyecto. Al recoger características de ambos juegos, el nicho de competidores aumentaría, pero este juego se diferenciaría lo suficiente de ambos como para poder encontrar su hueco en un mercado ya ocupado pero listo para explotar.

2.2. Propuesta

Como propuesta, se ha decidió crear un videojuego de cartas al estilo de ***Keyforge***, con un sistema de combate basado en una mezcla entra las características de ***Dungeons & Dragons*** y con un combate automático al estilo del ***Teamfight Tactics***. Esta es una propuesta original, ya que ningún juego visto en el análisis del mercado se centra como este proyecto en esta modalidad de juego. Si es cierto que hay varios juegos que mezclan ambos modos, como el ***Hearthstone Battlegrounds***, aunque es solo un modo y no permite la creación de mazos única para este. Por estas razones se puede decir que este proyecto es un juego innovador en su campo.

3. Análisis del problema

En este corto apartado, se hablará sobre el marco legal de este proyecto, sobre todo a la hora de usar ciertos elementos visuales.

3.1. Análisis del marco legal y ético

3.1.1.Propiedad intelectual

Para el uso de ciertos elementos del estilo gráfico, se ha hecho uso de varios paquetes de recursos recogidos en itch.io. Entre ellos destacan:

- **Para el uso de menús y botones**: *Wooden Pixel Art GUI*, de **Narik**. [20]
- **Para algunas animaciones**: *Fantasy Card Assets*, de **CafeDraw**. [21]
- **Para los ataques**: *Fantasy Icon Pack*, de **Ravenmore**. [22]

Estos paquetes están disponibles tanto para el uso comercial como para el no comercial.

Por otra parte, se ha hecho uso de imágenes con derecho de autor en ciertos elementos de la partida, de los que se hablará a continuación. Esto se ha decidido hacer de esta manera ya que este proyecto realmente se trata de un sistema de combate, con un modo de juego para probarlo. Por esa razón, este proyecto realmente nunca se comercializará o saldrá al mercado, sino que se usarán sus tecnologías, implementadas dentro del proyecto de Enric Puigcerver.

- **Para el menú principal**: Imagen de portada del Manual de monstruos de Dungeons & Dragons.
- **Para el fondo de la partida**: Arena base del videojuego Teamfight Tactics, de Riot Games.

4. Diseño de la solución

En este apartado se va a ver y comentar las reglas y el diseño del arte de la partida, de esta manera se entenderá el objetivo y jugabilidad del videojuego, que servirá más adelante a la hora de explicar por qué se han escogido las tecnologías que se usan y las razones tras la programación de este videojuego. Primero, se empezará con las reglas, centrándose en cómo se desarrolla una partida y los tipos de carta y sus atributos.

Más adelante, se hará hincapié en las tecnologías escogidas, explicando por qué se han elegido estas en concreto y cómo se han usado. Para finalizar, poco a poco se irá elaborando el desarrollo de este proyecto de una forma más detallada, explicando paso a paso cómo funciona el mismo.

4.1. Reglas

4.1.1. Partida

Una partida de este videojuego se trata de un combate de cartas uno contra uno por rondas. El objetivo final del videojuego es lograr la victoria de la partida al mejor de tres rondas, combatiendo contra la máquina. Al mezclar elementos comunes de los juegos de cartas y los auto-battlers, este juego se caracteriza por el manejo de colecciones de mazos, los cuales se han de elegir previamente a las batallas, así como una batalla automatizada de las cartas dentro del tablero.

Los mazos del usuario se deberán crear previamente a la partida, vinculándose automáticamente a la cuenta del usuario jugador. Un mazo se compone de diez cartas, los valores de sus atributos se crean de forma automática. La aleatoriedad a la hora de crear estas cartas y la cantidad de atributos de los que constan hacen que sea prácticamente imposible que existan dos cartas iguales en el juego y, por tanto, cada mazo sería único.

Una vez elegido el mazo para jugar, se empieza la partida. El jugador empieza con cinco cartas en su mano, las otras cinco guardadas en el mazo. El primer turno corresponde al jugador, pudiendo elegir qué cartas se desean sacar para jugar en esta ronda. Para poder jugar las cartas, estas han de cumplir el coste de maná, el cual vendrá dado por el atributo de nivel de las cartas.

En un principio, tanto el jugador como la máquina empieza con tres puntos de maná, que irá perdiendo a medida que juegue cartas, en un grado proporcional al valor del coste de maná de las cartas jugadas. Al principio de las siguientes rondas, el maná se restablecerá incrementando su valor por un punto. Al final del juego, un jugador podrá tener como máximo seis puntos de maná.

Al jugar las cartas, se habrá de interactuar con un botón indicando que se han dispuesto en el tablero y el jugador está preparado, por tanto, pasará a ser el turno del enemigo, la

máquina. De forma completamente aleatoria, la máquina irá depositando cartas hasta quedarse sin maná.

Cuando el enemigo deposite todas las cartas que vaya a jugar en el tablero empezará el combate. El combate se trata de una sucesión de rondas, donde cada turno se trata de la resolución de las acciones de una carta, por orden de una prioridad establecida por los atributos de las cartas.

Una ronda de combate acaba cuando uno de los jugadores, ya sea el usuario o la máquina, acaba sin cartas aliadas en el tablero. Cuando esto suceda, el jugador que siga con cartas en el tablero se declarará vencedor de la ronda.

Cuando se gane una ronda, las cartas del jugador victorioso volverán a su mano, con su atributo de vida actual, pero incrementando el valor de algunos atributos dependiendo de la clase a la que pertenezca.

A partir de esta segunda ronda, se habrá recuperado el maná perdido y se podrá usar el mazo correspondiente para robar cartas. Se podrá robar del mazo hasta volver a tener cinco cartas en la mano, ya que es el máximo.

De forma continuada, se irán repitiendo las rondas de combate sin cesar de esta manera hasta escoger un vencedor al mejor de tres. Cuando esto suceda, se establecerá la puntuación correspondiente al perfil de usuario del jugador y se habrá acabado la partida.

4.1.2.Cartas

Uno de los elementos centrales de este videojuego son las cartas, en este punto se van a describir los tipos de carta y los atributos que componen a cada uno. En este videojuego, una carta es la unidad mínima que se necesita para empezar un combate, esta es una mezcla de atributos con valores creados de forma aleatoria que representan alguna criatura, objeto o hechizo.

Se va a empezar por explicar las cartas de tipo criatura, ya que son el tipo de carta más importante, o principal, dentro del juego. Una carta de este tipo representa, como su nombre indica, a una criatura. Los atributos de las cartas están basados en el juego de rol ***Dungeons & Dragons***. [23]

Uno de los atributos principales de una criatura es su **raza**, la cual otorga una base a atributos como son la fuerza, la destreza, la constitución o la afinidad a la magia que tiene esa criatura. Otro de los atributos más importantes es la **clase** de la criatura. Esta clase otorga atributos como son la vida o las armas que pueden llevar.

Continuando con los atributos principales siguen las **armas** y la **armadura**. El arma de la criatura proporciona el máximo de daño base que puede realizar esta. La armadura, en cambio, representa el umbral de una criatura para tener que recibir el daño entrante. Este umbral solo se aplica a ataques de otras criaturas, y no a otro tipo de daño que puede recibir, aunque esto se verá más adelante. También cabe recalcar que una armadura puede venir acompañada de un escudo, otorgándole puntos de armadura extra, aunque para que se de esta situación ha de ser

un caso especial donde la clase y el arma de la criatura lo permitan. Por ejemplo, una criatura cuya arma fuese un arco largo y no tuviese armadura, no podría estar sujetando un escudo.

Ya se han visto los cuatro atributos principales, a continuación, se va a describir el resto de los atributos pertenecientes a las cartas, los valores de los cuales vienen dados en parte por mezcla de los atributos principales junto con cierto componente aleatorio.

El **nivel** de una carta indica lo poderosa que es esta en potencia. Al ser en gran parte aleatorios los atributos, el nivel no es indicativo de superioridad en comparación con una carta de menor nivel, pero este atributo si llega a otorgar cierto incremento de otros atributos llegando así a crear cartas potencialmente mejores. Este nivel también se corresponde al coste de **maná** de una carta, ya que una carta de mayor nivel suele ser mejor que una carta de menor nivel.

La **vida** de una carta son los puntos que indican cuán difícil de vencer es esta. A mayor vida tenga una criatura, mayor dificultad para acabar con ella. La vida viene dada por la clase de la criatura, en conjunto con el nivel y la constitución de esta, que se verá a continuación.

La **constitución** de una criatura indica lo resistente que es esta. Este atributo viene dado por la raza y solo sirve realmente como apoyo al atributo de vida. De forma similar funcionan los atributos de fuerza, destreza y magia, que se verá a continuación.

La **fuerza** es el atributo correspondiente a la fuerza física de una raza, por eso, este valor viene dado por el atributo de raza. Esta fuerza determina el incremento a los ataques de las criaturas con tipo de daño físico.

La **destreza** es lo veloz, o precisa, que es una criatura. Al igual que los atributos anteriores, viene dado por la raza. La destreza sirve para aumentar la armadura de una criatura, ya que representaría la probabilidad de esquivar un golpe. También se cuenta la destreza para elegir la **prioridad** de una carta, ya que una criatura más rápida atacaría antes que una lenta.

La **magia**, o afinidad mágica, representa el poder mágico que tiene la criatura en cuestión. Este valor suele ser cuantioso en las criaturas que golpean con un tipo de daño mágico, ya que este tipo de criaturas suelen escasear en otros atributos como la destreza o la vida.

El **rango** de una carta viene dado por el arma que porte. No es lo mismo una criatura que sujete una espada que una que sujete un arco. Este atributo influye a la hora de atacar, ya que una carta con mayor rango no necesitará acercarse al lado de las criaturas objetivo para golpearlas.

La **posición** simboliza en que celda del tablero se encuentran. Este atributo no influye en ningún otro rasgo, pero sirve para saber dónde se encuentra cada carta en cada momento de la partida.

4.2. Tecnologías utilizadas

En este punto se van a definir algunas de las tecnologías o herramientas utilizadas dentro del proyecto, así como otras definiciones necesarias para su comprensión. También se verán qué tecnologías son usadas para la creación de este videojuego, para así tener más entendimiento sobre que se ha usado y por qué se ha hecho de esa manera, y tener una idea general para apartados más avanzados.

4.2.1.Definiciones

Un **videojuego casual** es un juego de corta duración, apto para todo tipo de jugadores, tanto jugadores más experimentados como jugadores casuales. Este tipo de juegos suele hacerse popular en entornos de descanso, ya que su corta duración permite al usuario descansar la mente durante unos instantes sin la necesidad de requerir de toda su atención durante un largo periodo de tiempo. Esta clase de videojuego tampoco castiga a aquellos que pasan grandes cantidades tiempo sin jugar, ya que su sencillez permite al usuario volver en cualquier momento preparado para jugar.

Una **ontología** es un conjunto de tipos (clases) y propiedades, así como una relación entre estos. Para poder trabajar con facilidad con estos tipos se creó **OWL**, acrónimo del inglés ***Web Ontology Language***, este es el lenguaje que se va a usar para acceder a la ontología. Una ontología cataloga las variables necesarias para algún conjunto y establece la relación entre estas. En este videojuego, la ontología será la base tanto para guardar lo relativo al usuario y a la colección de mazos y cartas, como para acceder a los atributos de las cartas.

Un **agente inteligente** es un programa el cual puede percibir su entorno y tiene la capacidad de interactuar conforme a la información recibida de este y decidir qué acciones tomar mediante un sistema de evaluación para llegar a un objetivo final el cual persigue. En este videojuego, cada carta sería un agente inteligente, cuyas propiedades son dadas por la ontología, y los cuales perciben de su entorno, que en este caso sería el tablero, para entonces razonar cuál sería su mejor comportamiento.

Un MAS (***Multi-Agent System***, o Sistema Multi-Agente) es un sistema compuesto por múltiples agentes, los cuales se comunican entre ellos para obtener información de su entorno y así, poder cumplir con sus objetivos. Ahora que se entiende qué es un agente, se puede ver que, al tener varias cartas en una partida, es decir, varios agentes, y que estos se comuniquen entre ellos, una partida de este videojuego lo conforma un Sistema Multi-Agente, comandado por un agente “principal”, que se encarga de decidir y actualizar la información de la partida en todo momento. Para la comunicación de estos agentes se necesitará hacer uso de un servidor XMPP, del cual se hablará más adelante.

Un **socket** es un canal de comunicación formado por la combinación de una dirección IP de origen y destino y los números de puerto relativos a estas. En este proyecto se han usado sockets de tipo TCP.

TCP (*Transmision Control Protocol*, o Protocolo de Control de Transmisión) es un protocolo de transmisión de datos seguro, ya que se encarga de confirmar una autorización entre cliente y servidor antes de empezar con el intercambio de datos.

Para transmitir los mensajes entre el manejador de ontologías y agentes con el render del videojuego, se han usado usado las bibliotecas de socket necesarias en cada programa, creando servidores diferentes puertos de localhost. Más adelante en el trabajo se hablará con más detalle de las conexiones vía socket.

4.2.2.Tecnologías

Protégé es un entorno de desarrollo de ontologías **OWL**. Este ofrece una interfaz intuitiva y cómoda para el manejo de ontologías, ofreciendo además características funcionales muy interesantes como la posibilidad de usar un razonador, o *reasoner*, para inferir sobre las clases y propiedades. Aunque en este proyecto no se haya hecho uso de esta característica, podría llegar a ser una función con mucho potencial en un futuro en esta aplicación. [2]

En el proyecto se ha optado por **Protégé** debido a su sencillez a la hora de desarrollar las ontologías. Gracias a su simple interfaz se ha podido extender la ontología seleccionada y añadir tanto nuevas clases como propiedades de forma rápida y clara.

OwlReady2 es una librería y extensión de Python que sirve para la creación y gestión de ontologías. Esta permite una amplia variedad de funciones tanto para recorrer y buscar elementos de ontologías como para crear o editar nuevas clases o propiedades. [4]

En este trabajo, se ha usado esta librería para la creación de usuarios, mazos y cartas de este videojuego. A simple vista, se han creado individuos haciendo uso de las clases y propiedades extendidas de la ontología, más adelante se desarrollará ampliamente el funcionamiento de esta útil extensión.

SPADE (*Smart Python multi-Agent Development Environment*, o Entorno de Desarrollo de Sistemas Multi-Agente Inteligentes en Python) es una plataforma de desarrollo de MAS basada en la mensajería instantánea. [6]

Este tiene métodos para la creación rápida de agentes, los cuales van vinculados a un comportamiento, o *behaviour*. Los comportamientos representan la forma en la que actúa y ejecuta las acciones el agente. Un agente puede tener uno o varios comportamientos, depende de la funcionalidad que se le quiera dar.

Para los agentes que utiliza este proyecto, tanto el Manager como las cartas, se ha decidido usar un comportamiento de máquinas de estados finitos, o **FSMBehvaiour**. Esta clase de estado, como su nombre indica, se comporta como una máquina de estados, donde cada estado es un comportamiento de clase **OneShotBehaviour**, es decir, se ejecuta tan solo una vez y finaliza. Se ha elegido esta clase de comportamiento ya que simula el avance de las fases de un videojuego, más adelante se entrará en detalle sobre este tema.

Para iniciar un agente se debe, primero que nada, crear su instancia, y luego conectarlo a un servidor XMPP. Este servidor XMPP es el encargado de realizar todas las comunicaciones internas y entre agentes, así como de autenticarlos.

Un **servidor XMPP** (*Extensible Messaging and Presence Protocol*, o Protocolo extensible de mensajería y comunicación de presencia) es un sistema de mensajería instantánea basado en XML. [19]

Este servidor se usa para conectar los agentes de este proyecto, con el fin de permitir que se comuniquen entre ellos. Estos agentes funcionarían como clientes del servidor. Como servidor se ha decidido escoger **lightwitch.org**, por su facilidad a la hora de crear las conexiones necesarias entre agentes y servidor.

Unity es un motor de videojuegos tanto 2D como 3D, con una amplia cantidad de herramientas y funcionalidades para la creación y desarrollo de videojuegos. En este videojuego

se ha usado Unity tan solo como render, es decir, para representar gráficamente el avance de la partida que llevan a cabo los agentes. [8]

No solo se ha usado Unity para representar el estado de las partidas, sino también para poder interactuar con la partida a la hora de realizar acciones como jugar las cartas o interactuar con los botones dentro del juego.

Para finalizar con Unity, se ha creado también con este motor un menú principal con una interesante selección de menús y vistas, como las referentes a la gestión de usuarios y la gestión de colecciones de un usuario.

Para desarrollar el proyecto se han usado varios lenguajes de programación y la coordinación entre estos a través de mensajería a través de varios sockets. Por una parte, el desarrollo de los agentes se ha programado con el lenguaje Python, mientras que el desarrollo dentro del motor gráfico de Unity se ha hecho con el lenguaje C#.

Python es un lenguaje de programación multiparadigma, con aspecto de los lenguajes orientados a objetos, de programación imperativa y funcional. Una de sus principales características es el uso de tipos dinámicos, permitiendo a las variables tomar distintos valores de tipos. Otra característica importante de este lenguaje es la facilidad de extensión, pudiendo así crear y desarrollar una amplia variedad de nuevos módulos para extender sus funcionalidades. [15]

Dicho esto, se usa Python para la gestión de la ontología con la librería **OwlReady2**, y para el desarrollo de los agentes mediante la extensión de **SPADE**, de las que ya se ha hablado en este apartado.

C# es un lenguaje de programación multiparadigma, al igual que Python. A diferencia de este, C# si usa un tipado en sus variables y objetos, permitiendo menos flexibilidad a la hora de escribir, pero supliéndolo con una mayor consistencia. Se ha optado por este lenguaje de programación ya que es el lenguaje que usa Unity a la hora de trabajar con scripts. [10]

4.3. Arquitectura del sistema

En esta sección se va a hablar de una forma más extensa y desarrollada de las tecnologías vistas en el apartado anterior y sus relaciones entre ellas. Al finalizar este apartado se pretende que el lector tenga una visión clara de cómo funciona cada componente no solo de forma individual, sino como un gran proyecto en su totalidad, se comprenderán las ideas detrás de la programación y la forma de desarrollar el proyecto.

Esta sección se va a acompañar en gran medida con diagramas UML de las clases del proyecto o los estados que puede tener la partida, para mejorar la comprensión de estas tecnologías como una unidad y no por separado.

4.3.1.Ontología

4.3.1.1. Arquitectura de la ontología

Para la ontología se ha optado por extender una ontología ya existente cuyo tema era el aclamado juego de rol ***Dungeons & Dragons***. En primera instancia se va a hablar de las clases ya existentes en la ontología, aunque tan solo de aquellas cuyo valor o uso en este proyecto tiene alguna importancia, y en el próximo punto se hablará de las extensiones que se han decidido llevar a cabo. A continuación, se van a recorrer las clases significativas de la ontología base:

- **CClass**, esta representa la clase, o rol, que desempeña una carta. Al mismo tiempo es una clase padre de numerosas clases cuyos nombres son específicamente el rol que desempeñan.

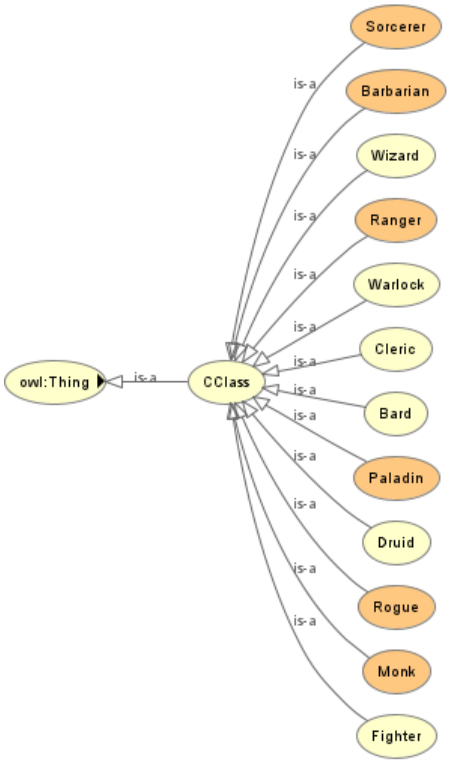

Figura 4.1: Grafo de la clase CClass

- **Race**, esta clase representa la raza de una carta. Esta clase, al igual que la nombrada anteriormente, es padre de un amplio número de subclases cuyos nombres son las razas a las que pertenecen.

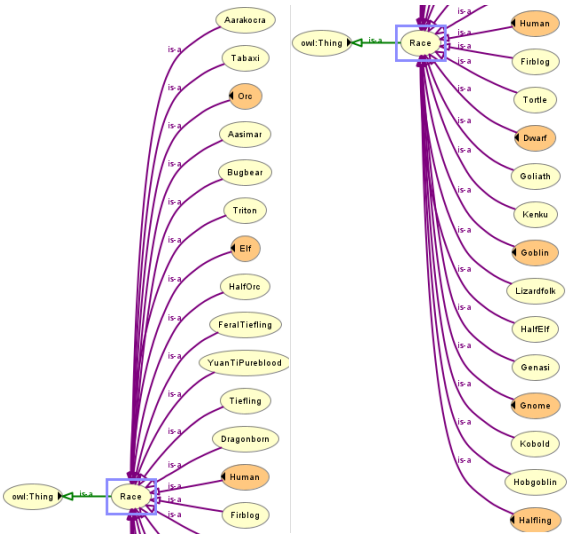

Figura 4.2: Grafo de la clase Race de la ontología

- **Item**, esta clase sirve como padre de numerosos objetos clave que pueden poseer las cartas, tales como la armadura o el arma que portan. Esta clase contiene una propiedad de datos de tipo entero que hace referencia a la durabilidad del objeto en cuestión. Al igual que las anteriores clases, es padre de una gran cantidad de hijos que especifican la clase de objeto que porta una carta.

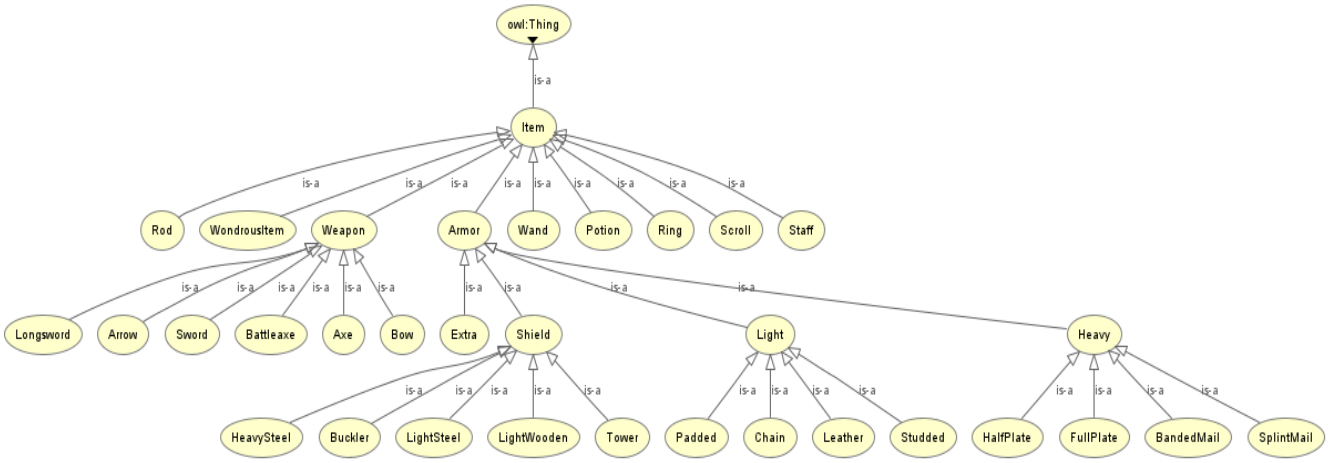

Figura 4.3: Grafo de la clase Item de la ontología

- **Spell**, en esta clase se pretende enseñar los distintos tipos de conjuros de los que se puede tener una carta, teniendo la clase Spell varias subclases las cuales representarían esta variedad de conjuros. Esta clase tendría entre propiedades de datos su rango y su daño, ambas propiedades siendo de tipo entero.

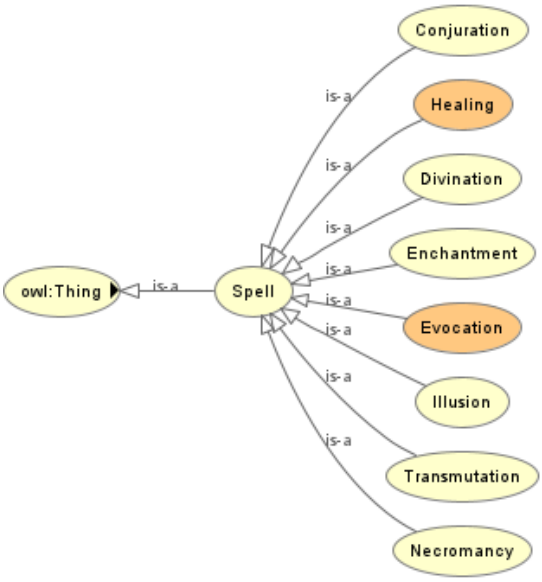

Figura 4.4: Grafo de la clase Spell de la ontología

4.3.1.2. Extensión de la ontología

Antes de empezar a hablar de la extensión realizada a la ontología que se ha escogido, se mostrará un diagrama general de cómo se ven las relaciones entre las nuevas clases, y así, poco a poco, explicar las nuevas clases o atributos que se han creado o modificado para entenderlos mejor.

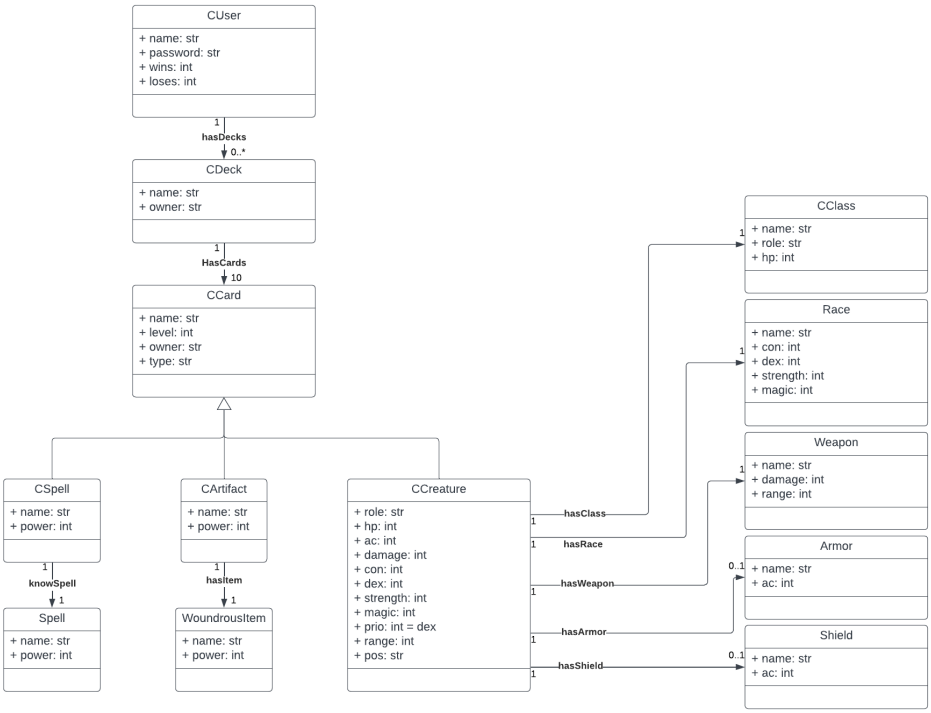

Figura 4.5: Diagrama de relaciones entre las clases de la ontología extendida

Para extender esta ontología a un juego de cartas, se ha empezado creando la clase más importante, **CCard**. Esta clase define una carta, con todas sus propiedades y atributos, las cuales se han tenido que definir también en la ontología.

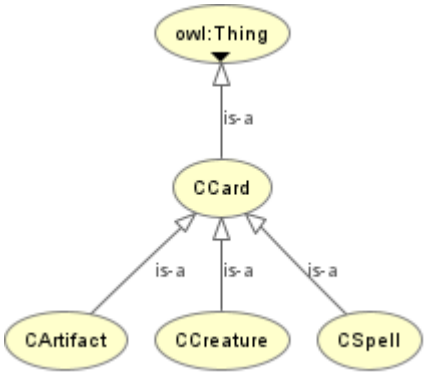

Figura 4.6: Grafo de la clase CCard de la ontología extendida

Primero se va a explicar cómo se ha definido una carta. Una carta se define principalmente por su Clase, **CClass**, y su Raza, **Race**, clases ya vistas con anterioridad, pero ahora extendidas. Estas dos propiedades les otorgarán sus principales atributos gracias a las nuevas propiedades de datos, como por ejemplo la vida, resistencias, fuerza y rapidez. Luego, una carta tendrá algún arma, la que representa el daño máximo que puede hacer una carta al atacar, y le dará un rango, pues no es lo mismo atacar con una espada que con un arco.

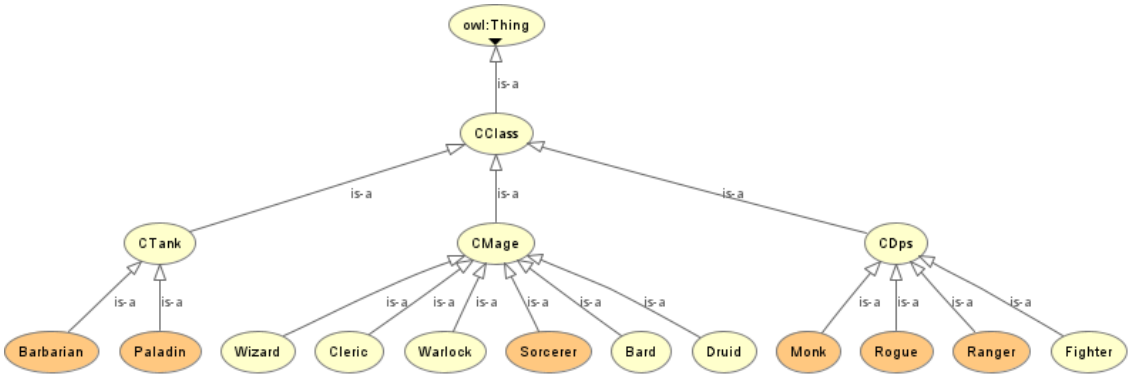

Figura 4.7: Grafo de la clase CClass de la ontología extendida

También la carta tendrá una armadura, que le dará la posibilidad de resistir o evadir un ataque enemigo, además cabe la posibilidad que una carta además de portar una armadura porte un escudo, otorgándole armadura extra, pero eso no es todo, ya que también es posible que una carta no lleve ninguna armadura ni escudo, haciendo así que solo mantenga la armadura base propia de la clase. Estos objetos vienen dados por las clases hijas de la clase padre **Item**, de la que ya se ha hablado en el punto anterior.

Los atributos de la carta vendrán dados por la combinación que le otorgan sus propiedades. Por ejemplo: la vida vendrá dada por el *hp* de la clase más la constitución de la

raza; el daño vendrá dado por la fuerza de la raza más el daño del arma; la armadura vendrá dada por la suma de la armadura y escudo, si es que lleva, junto con la destreza.

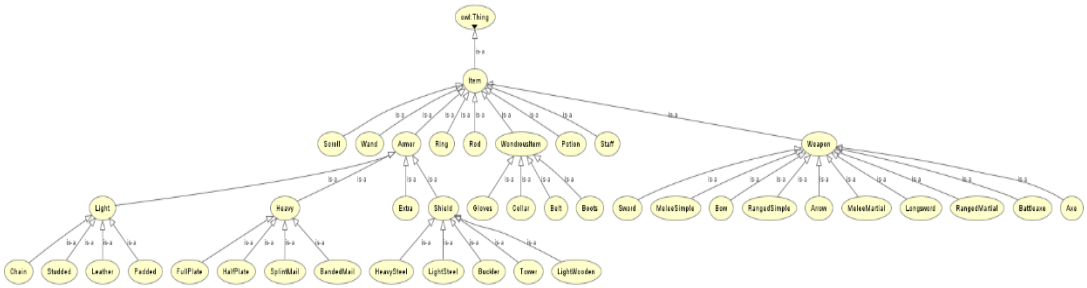

Figura 4.8: Grafo de la clase Item de la ontología extendida

Para darle un poco más de caracterización al estilo ***Dungeons & Dragons,*** se ha decidido dotar a algunos atributos un factor aleatorio, que simbolizaría la famosa "tirada de dados" de los juegos de rol. A continuación, se va a describir que compone una carta.

Ya se ha explicado qué es y cómo está definida una carta en nuestra ontología, pero para jugar a este juego primero se debe seleccionar un mazo o baraja, es decir, una combinación de varias cartas. Para crear este mazo en la ontología he creado una nueva clase llamada **CDeck**.

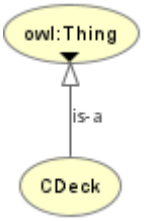

Figura 4.9: Grafo de la clase CDeck de la ontología extendida

Con estas definiciones ya se tiene la extensión necesaria de la ontología en cuanto a las cartas se refiere, aunque no es la extensión al completo, ya que se han añadido nuevas clases y propiedades específicamente para la gestión de usuarios de este videojuego.

Sobre la gestión de usuarios, se ha optado por la creación de la clase **CUser**, la cual define al usuario en sí. Un usuario lo componen su nombre de usuario, su contraseña, los mazos que tiene asociados y, por último, las victorias y derrotas de las que consta.

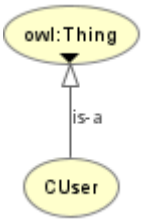

Figura 4.10: Grafo de la clase CUser de la ontología extendida

4.3.2.Agentes

4.3.2.1. Arquitectura de los agentes

En cuanto a los agentes, desde un principio se tenía claro como se iba a llevarlo a cabo, en un principio debería haber un agente controlador, al que se va a nombrar **Manager**, este se encargaría de tener en todo momento el estado de la partida y transmitirlo a las cartas, siendo estas también agentes.

También se tuvo claro el comportamiento que debería tener el Manager, un comportamiento de máquina de estados finitos, ya que este mantiene un comportamiento muy similar al desarrollo de un juego de cartas por turnos. A continuación, se va a ver como se desarrolla en un principio el comportamiento de este agente.

AgentManager es el nombre del programa y agente encargado del control de la partida. Este es el encargado de crear las conexiones de socket tipo TCP necesarias, tanto como servidor de escucha en la dirección IP de localhost y en el puerto 8001, como las conexiones como clientes TCP al servidor de escucha habilitado dentro de los scripts de Unity en la dirección IP de localhost y en el puerto 8000.

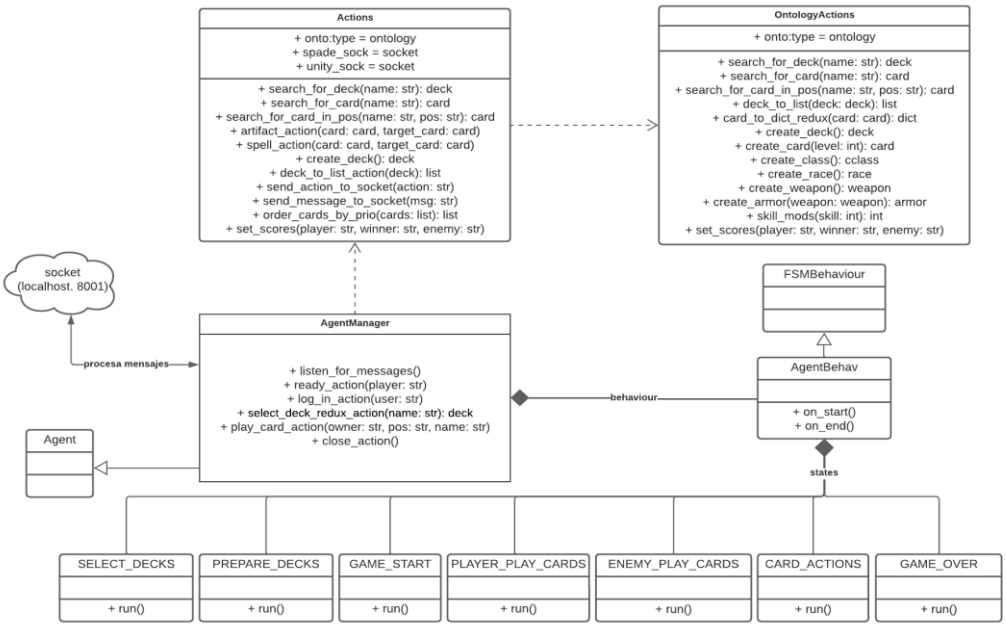

Figura 4.11: Diagrama UML de la gestión del AgentManager

AgentManager también se encarga de mantener ciertas variables clave para el estado de la partida, como pueden ser las listas con los agentes activos, los mazos de cada jugador, la puntuación de la partida o las posiciones del tablero libres y ocupadas.

Dentro de los estados que componen este agente se encuentran:

1.- **SELECT_ DECKS**:

Indica la señal al videojuego para empezar la partida mediante un mensaje y abre la conexión del socket de escucha esperando a recibir el mensaje del jugador.

2.- **PREPARE_DECKS**:

Crea un mazo aleatorio para el enemigo, inicializando las variables globales referente a los mazos en juego de este y, traduciendo a *json* el contenido de los mazos, es enviado a través del socket al videojuego para obtener así las cartas de este en el render.

3.- **GAME_START**:

De momento este estado solo sirve como un estado de transición entre los estados referentes a la elección y preparación de mazos y al desarrollo de la partida.

4.- **PLAYER_PLAY_CARDS**:

Este estado se encarga de activar la señal en el videojuego dando paso al jugador para poder jugar las cartas que desee. Cuando esto este hecho, se enviará la información necesaria a través de mensajes al estado, creando así los agentes (que representan a las cartas jugadas) necesarios, poniéndolos en espera.

5.- **ENEMY_PLAY_CARDS**:

Este estado es análogo a su estado predecesor, activa la señal del enemigo permitiendo que juegue sus cartas y, espera recibir un listado con las cartas para crear los agentes necesarios e instanciarlos en las listas correspondientes.

6.- **CARD_ACTIONS**:

Es a partir de este estado cuando realmente empieza la partida. Por una parte, ordena las listas de agentes por prioridades (estas prioridades están dadas dentro de las propiedades de la carta). Cuando está ordenado, se encarga de recorrer uno a uno cada agente dentro de la lista ordenada. Primero se encargaría de comprobar si alguno de los jugadores cumple con la condición de victoria necesaria para pasar al próximo turno, si fuese así, acabaría con el bucle de juego y pasaría a resolver estas acciones, si no lo fuese, iniciaría su comportamiento, que ya se verá más adelante.

A estos agentes, se les dota con el estado actual de la partida, para que así no haya ningún error durante la ejecución. Luego, espera a recibir un mensaje de terminación por parte del agente, volviendo a modificar las variables de estado con el nuevo estado al final la ejecución de este.

Finalmente, hace las comprobaciones necesarias para ver si continuar con la ejecución de los agentes. En el caso de que no haya ningún ganador transitaría a este propio estado CARD_ACTIONS. en el caso contrario, sumaría la puntuación necesaria dentro del estado de la partida y, empezaría una nueva ronda transitando al estado PLAYER_PLAY_CARDS. En el caso de cumplir con las condiciones de victoria necesarias al tener la puntuación correspondiente a la victoria, se añadiría dentro del usuario de la ontología el punto de victoria o derrota y transitaría al último estado GAME_OVER.

7.- **GAME_OVER**:

Indica cuando finaliza la partida y se encarga de todas las operaciones necesarias para que esto sea posible.

A continuación, se va a comentar sobre cuando transitaría cada estado, acompañado de un pequeño diagrama para visualizar estas transiciones como una máquina de estados finitos.

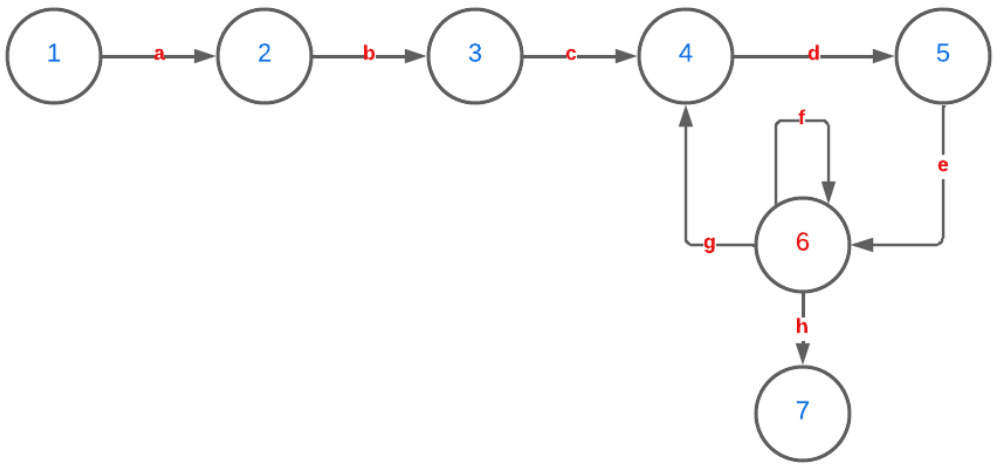

Figura 4.12: Diagrama de transiciones de la máquina de estados finitos del AgentManager

a) El Manager ha recibido y actualizado el mazo del jugador.
b) Se ha creado el mazo rival y se han enviado ambos mazos al videojuego.
c) Se ha hecho la transición necesaria para empezar el juego
d) El jugador ha jugado sus cartas y se han creado los agentes necesarios.
e) El enemigo ha jugado sus cartas y se han creado los agentes necesarios.
f) Se han iniciado todos los agentes y se han resuelto sus acciones, pero el juego no ha terminado.
g) Una de las rondas ha finalizado, estableciendo las puntuaciones necesarias y empezando una nueva ronda.
h) Uno de los jugadores cumplía con las condiciones necesarias de victoria.

Ya se ha visto el pensamiento y funcionamiento que hay tras el Manager. Ahora hay de ver que hay detrás de los verdaderos pilares del videojuego, las cartas. Para esto, se va a ver la arquitectura que se encuentra detrás de la clase **CardActions**.

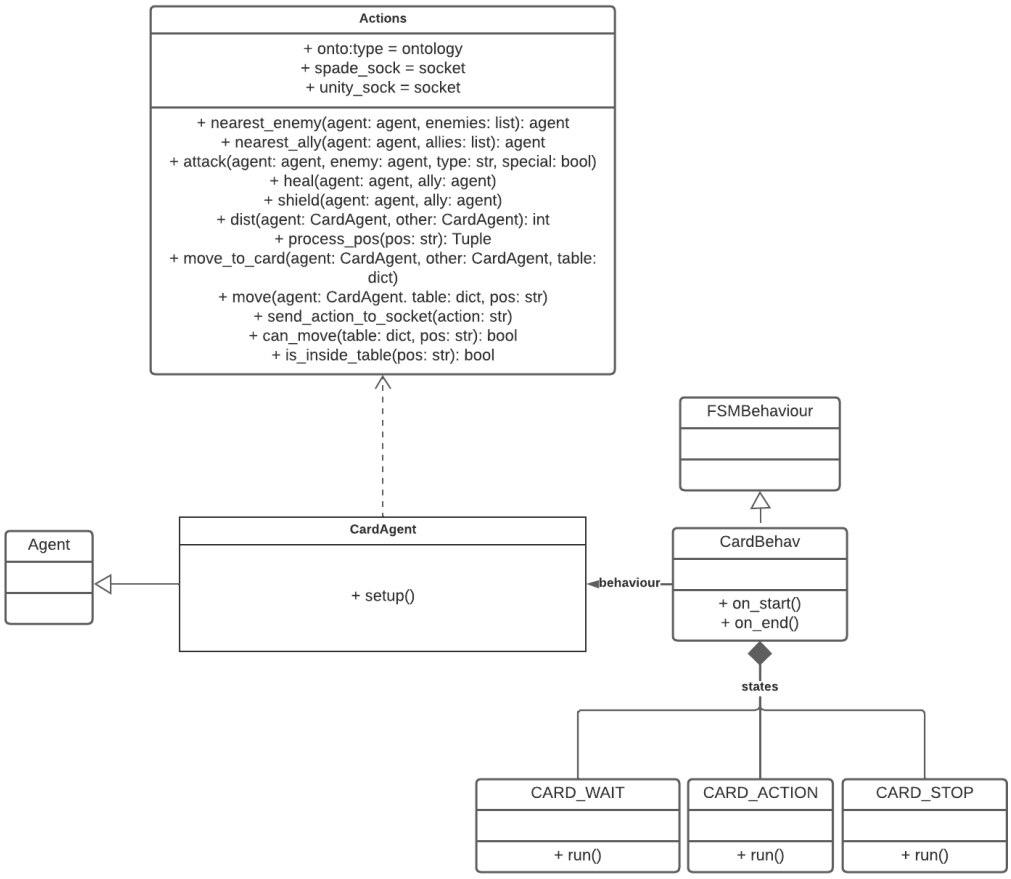

Figura 4.13: Diagrama UML de la gestión del CardAgent

A continuación, se verá cómo se escogió el método de creación de los agentes de las cartas. Por una parte, no se sabía cuándo iban a ser creados los agentes, si desde scripts externos nada más se jugase una carta, ya que al hacer esto no hubiese hecho falta la creación e inicialización de un socket de escucha dentro del Manager, o dentro de este mismo, con la creación de un socket de escucha. Finalmente, se optó por la creación de los agentes dentro del Manager, ya que de esta forma se podría tener mucho mejor control del estado desde el propio Manager, y no depender completamente de la transmisión con mensajes vía el servidor XMPP entre agentes carta y el agente Manager.

Ahora ya que se buscaba para que funcionase la creación de los agentes se tenía que pensar en cómo debía ser su comportamiento. En un principio se debatió entre dos opciones, una de ellas era un comportamiento cíclico, y otra fue, al igual que el Manager, un comportamiento de máquina de estados finitas.

Al final se optó por la segunda opción ya que, por una parte, el control que se habría de hacer al tener todos los agentes actuando de forma simultánea para que no sucediesen condiciones de carrera podría llevar a tiempos de espera entre acciones demasiado largos, haciendo que realmente no existiese el paralelismo que se buscaba y, por otra parte, que todas las cartas jugasen al mismo tiempo no simula realmente un combate en el principal juego referente, el ***Dungeons & Dragons***.

Con un comportamiento al estilo FSM, se podrían simular los combates por turnos tanto en los juegos de cartas, como en los juegos de rol. Decidido esto, quedaba decantarse por los estados que debería tener este comportamiento, entre los cuales se encuentran:

1.- **CARD_WAIT**:

Se encarga de esperar a recibir la señal necesaria, enviada a través del servidor XMPP por parte del agente Manager, para empezar a ejecutar las acciones propias de esta carta.

2.- **CARD_ACTION**:

Este estado es la base del videojuego, en este, la carta se encarga de ejecutar las acciones necesarias para combatir a sus enemigas o apoyar a sus aliadas. Dependiendo completamente de las propiedades que posee la carta, como es su clase para activar habilidades especiales como ataques poderosos o habilidades de apoyo con sus compañeros, así como para moverse.

Estas habilidades dependerán de los atributos que posea la carta, dados por la creación de estas en la ontología, así como de un factor aleatorio que simula una tirada de dados en los famosos juegos de rol.

Finalmente, al resolver todas las acciones posibles, transitará al siguiente y último estado.

3.- **CARD_STOP**:

Envía a través del servidor XMPP un mensaje al agente Manager, activando las señales necesarias para poder continuar con el siguiente agente.

A continuación, hay un pequeño esquema que permite visualizar la transición entre estados del comportamiento propio de los agentes de las cartas.

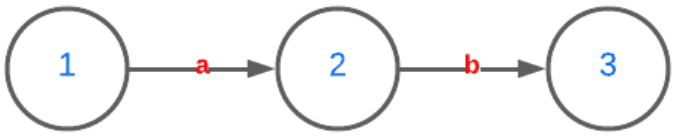

Figura 4.14: Diagrama de transiciones de la máquina finita de estados del CardAgent

a) Se ha recibido la señal para empezar a ejecutar las acciones de la carta.
b) Se han resuelto todas las acciones de la carta.

Con esto se puede dar por finalizada la arquitectura y el pensamiento que se encuentra detrás de los agentes.

4.3.3.Unity

4.3.3.1. Arquitectura del menú principal

Ya se ha recorrido la arquitectura y gestión detrás de la ontología y los agentes inteligentes, pero con tan solo estas tecnologías no se puede jugar una partida. Para jugar al videojuego se necesita una interfaz con la que interactuar y elegir, con la que crear los usuarios y empezar una partida, y para todo esto se ha usado el motor gráfico de Unity.

En esta sección se va a explicar como se ha decidido usar los componentes que ofrece Unity para crear el videojuego, la estructura interna que se le ha dado y cómo es posible que pueda conectarse con tecnologías tan diferentes y alejadas como son la ontología y los agentes inteligentes. Para ello, se irá recorriendo la cara del videojuego con imágenes y explicaciones de las ideas a la hora de programar. Para empezar, se mostrará la primera escena creada en Unity, la referente al menú principal. [13]

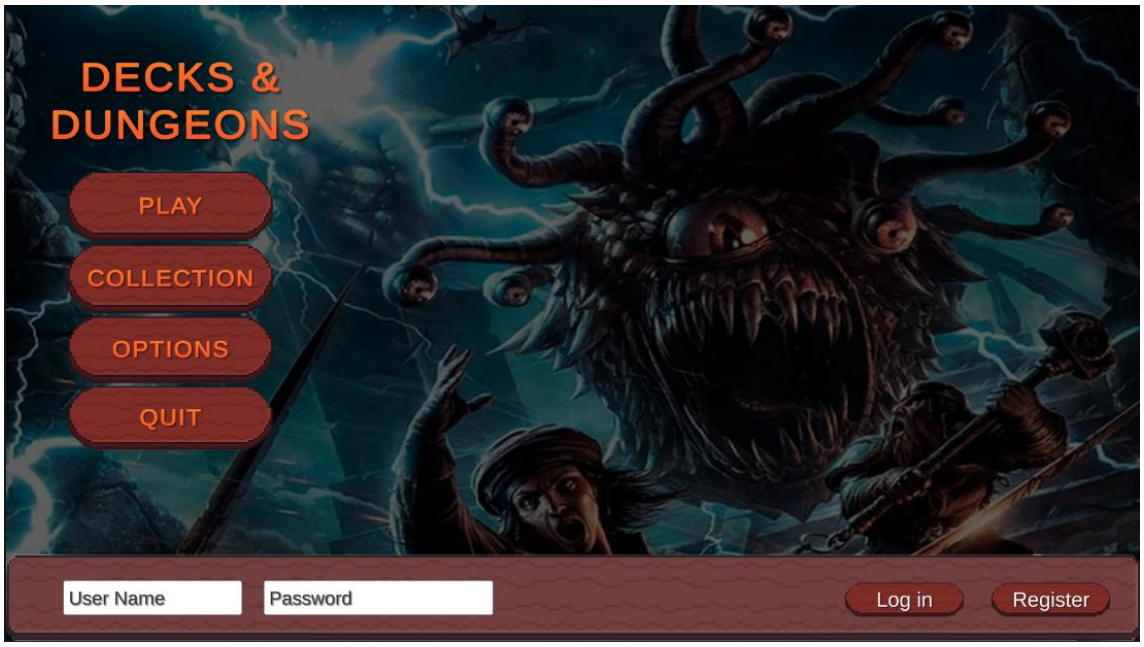

Figura 4.15: Imagen del menú principal

Este menú principal está creado sobre un objeto de *UI Canvas*, para poder hablar de este menú de forma correcta, se mostrará a continuación la estructura de los scripts que lo conforman, donde por una parte existe un script el cual sirve para interactuar con los objetos dentro de la pantalla y, por otra parte, un script que se encarga de crear las conexiones al socket de escucha (referente a la ontología) necesarias.

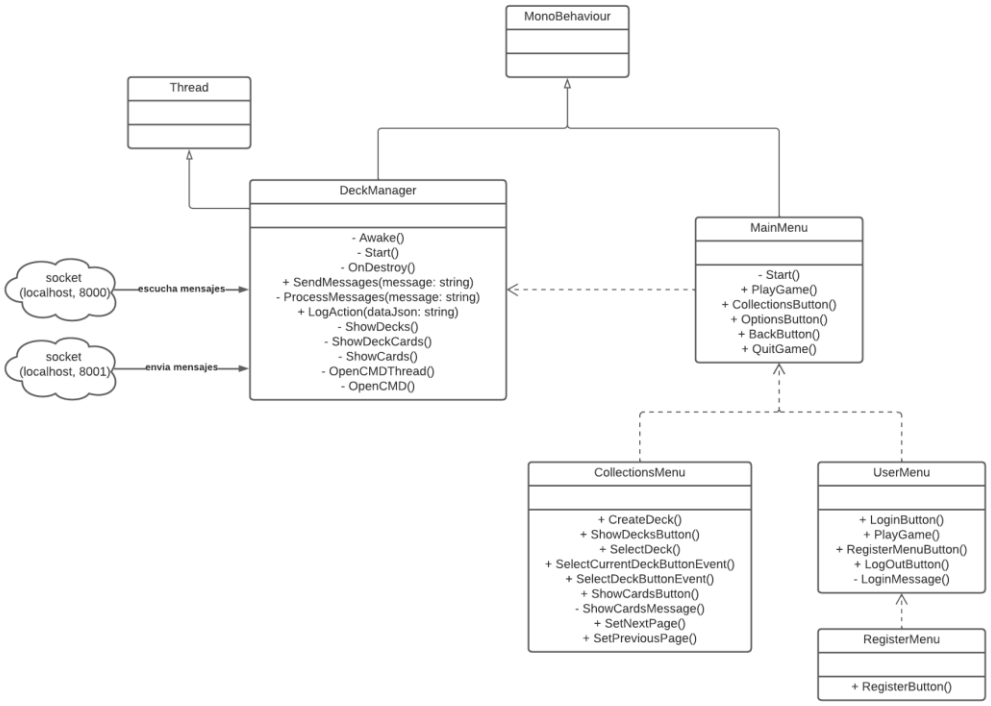

Figura 4.16: Diagrama UML inicial del menú principal

Como se puede observar, el script llamado **DeckManager** es el encargado de realizar las conexiones con sockets, así como enviar y procesar los mensajes. Por otro lado, el script **MainMenu** se encarga de controlar todo lo referente a lo que su nombre indica, el menú principal. A este script se le agregan varias otras clases, que se encargan de controlar los menús a los que su nombre se refiere, entre ellos el menú de colecciones, el de usuario y el de registro.

El script **DeckManager** es el encargado de iniciar el programa encargado de acceder a la ontología, es decir, el encargado de iniciar el script de nombre **OwlManager**. Para hacer esto, se ha decidido crear un *sub-thread* donde, en segundo plano, inicializa una terminal y ejecuta dicho script, iniciando así el socket servidor de escucha en el puerto 8002.

Figura 4.17: Diagrama del inicio de OwlManager

Mediante la interacción con los botones vistos en la pantalla del menú principal, se envían mensajes a través del **DeckManager**, conectándose como cliente al socket iniciado desde **OwlManager**, donde posteriormente este los procesa y envía su respuesta.

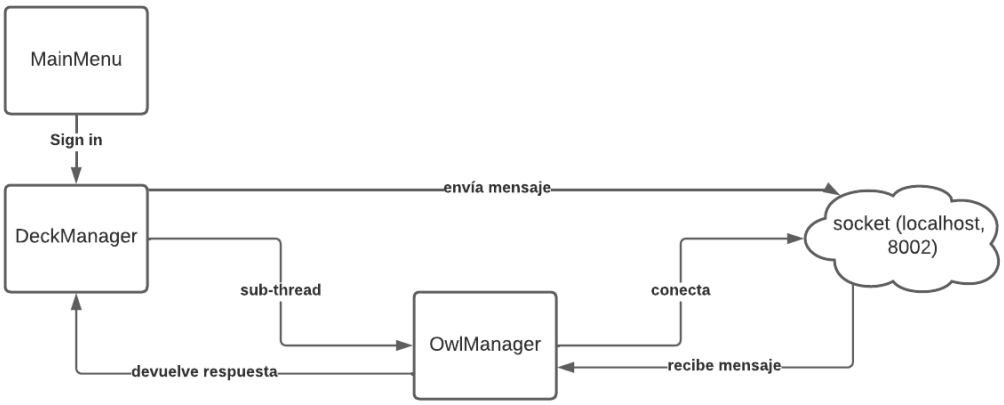

Figura 4.18: Diagrama básico del registro en el videojuego

En este videojuego, para empezar a interactuar con botones has de iniciar sesión. Si no se tuviese una cuenta se debería crear una desde el formulario correspondiente. Para hacer eso, se debería clicar el botón *Sign In*, el cual sirve para abrir el formulario de registro.

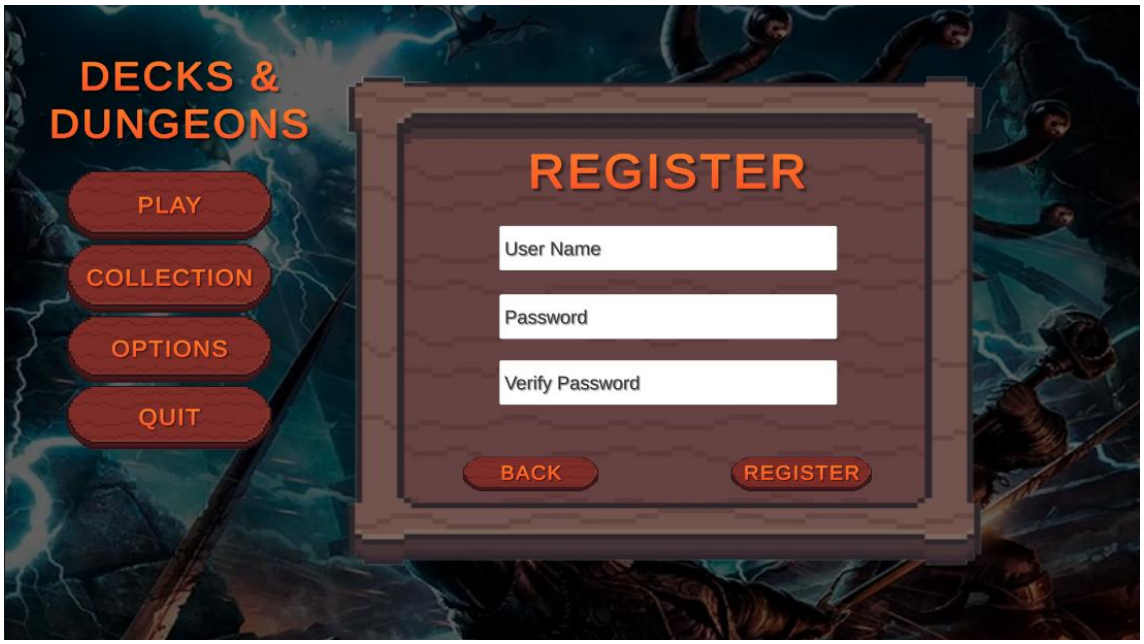

Figura 4.19: Imagen del menú de registro

Para proceder al registro se debe introducir un nombre de usuario y una contraseña. Para mayor seguridad, se debe repetir el campo de contraseña y verificar que ambas coinciden, en el

caso donde no coincidiesen ambas contraseñas, no dejaría crear la cuenta. En caso de coincidir, se volvería al menú principal, ahora con la sesión iniciada en la nueva cuenta.

Para crear la cuenta, **DeckManager** se conecta como cliente al servidor de escucha correspondiente enviando un mensaje con el nuevo nombre y contraseña. El socket recibe el mensaje y procesa su correspondiente acción, accediendo a la ontología y comprobando si existe tal usuario. Si fuese así, no se permitiría crear la cuenta, en caso contrario, añadiría la nueva cuenta a la ontología.

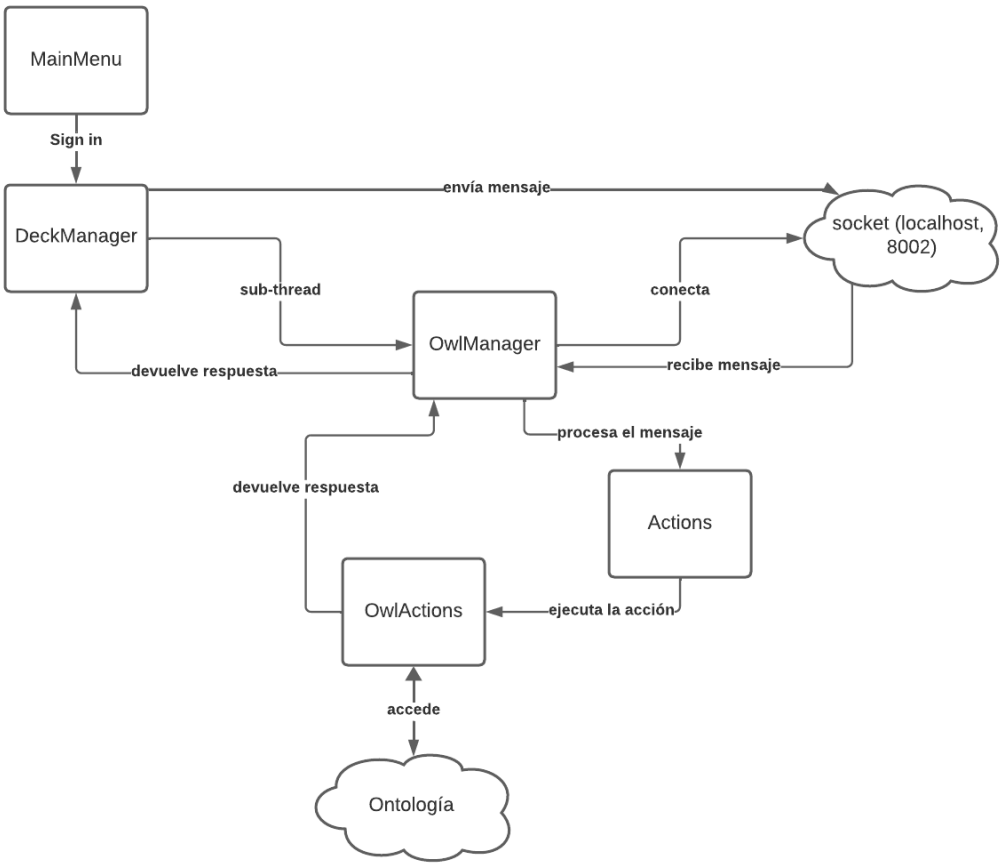

Figura 4.20: Diagrama de un ejemplo al acceder a la ontología en el inicio de sesión en el videojuego

Si todo ha ido bien y el usuario se ha registrado sin problema, en ese caso el videojuego iniciaría sesión de forma automática, sin necesidad de hacerlo el mismo usuario. Sin embargo, y a pesar de la similitud del procedimiento, se explicará a continuación como se daría el inicio de sesión desde el menú principal.

Para iniciar sesión se debe introducir el nombre de usuario y contraseña en los campos donde son requeridos. Posteriormente, al pulsar sobre el botón de inicio de sesión, se enviarán los mensajes correspondientes al socket y este los procesará como debe. En este caso, accedería a la ontología comprobando que existe el usuario introducido y le corresponde la contraseña dada. En caso de coincidir ambos campos, se habría iniciado sesión correctamente, y se abriría un nuevo menú de usuario, en caso contrario, no sería posible iniciar sesión en el juego.

Figura 4.21: Imagen del menú de usuario

Como se puede observar en esta pantalla se ve simple vista que el menú inferior para iniciar sesión se ha intercambiado por un menú lateral con información sobre el perfil del usuario. En esta se mostraría el nombre elegido por el usuario, el mazo actual seleccionado, es decir, con el cual se jugarán las partidas y las victorias y derrotas del perfil.

En un principio, al crear un usuario por primera vez, no se cuenta con ningún mazo y, por tanto, tampoco se tiene ningún mazo seleccionado. También cabe decir que tanto victorias como derrotas tienen un valor inicial igual a cero, aumentando este valor tras cada partida conforme el resultado final de estas.

En este momento tan solo se puede elegir entre, o cerrar la sesión del perfil, o acceder a la colección del usuario. Realmente se puede cerrar sesión en cualquier momento, siempre que sea fuera de una partida. Si se quisiera cerrar la sesión, se ocultaría este nuevo menú y se volvería a activar el menú principal.

Es importante saber que, para mantener los datos del usuario recordados para la próxima sesión de juego, se han creado varias variables de clase *PlayerPrefs*. Estas variables se guardarían en un registro local entre sesiones, para hacer más rápido el acceso a el usuario y sus datos en todo momento y no tener que recurrir al uso innecesario de mensajes. Entre los *PlayerPrefs* que se han decidido crear se encuentran: ***User***, referente al nombre usuario; ***SelectedDeck***, el cual indica el último mazo que ha sido seleccionado como activo; ***Wins*** y ***Loses***, ambos valores indicativos de las victorias o derrotas del usuario.

Antes de empezar a jugar se deberá seleccionar un mazo. En caso de que se intente iniciar una partida con el botón ***Play*** sin ningún mazo elegido, no se permitiría. Para eso se debe acceder antes a la sección ***Collection***, donde se encuentran los formularios que permiten interactuar con la colección de mazos y cartas asociada al usuario.

Figura 4.22: Imagen del menú de colecciones

Como en un principio el usuario no tiene ningún mazo asociado, se empezará enseñando la funcionalidad del botón ***Create Deck***.

Al usar el botón que permite crear un mazo, instantáneamente se enviaría un mensaje al servidor de escucha en el socket de **OwlManager**, escuchando en localhost con el puerto 8002. Al recibir el mensaje, el socket se encargaría de procesarlo, creando un nuevo mazo de cartas totalmente aleatorio y vinculándolo al usuario el cual ha invocado el método.

Ahora el usuario ya cuenta con su primer mazo, pero sigue sin poder jugar, para ello debe seleccionar el mazo con el cual quiere jugar. Para hacer esto se debe pulsar sobre el botón ***My Decks***, el cual se encarga de mostrar una lista con todos los mazos disponibles del usuario.

Figura 4.23: Imagen del menú de mazos

En esta nueva pantalla, se muestra un listado con el nombre de todos los mazos de los que dispone la cuenta, si se selecciona cualquiera de ellos aparecerá un nuevo botón llamado ***Select Deck*** que, como su nombre indica, permite seleccionar este mazo como mazo activo en la cuenta, y ya se podría jugar.

Al mismo tiempo, al seleccionar un mazo también se abre una nueva ventana. Esta ventana muestra el nombre de todas las cartas pertenecientes al mazo. Esta funcionalidad se resuelve enviando un mensaje al socket con tan solo el nombre del mazo como dato. Esto permite al script correspondiente acceder a la ontología para recuperar ese mazo y listar todas las cartas pertenecientes a él, devolviéndolas como respuesta del mensaje.

Figura 4.24: Imagen del menú de mazos extendido

El botón encargado de seleccionar mazo se encarga de modificar, o crear si no existe aún, la variable de *PlayerPrefs* referente al mazo seleccionado. Al finalizar este método, debería existir una variable guardada en recursos locales cuyo valor sea el nombre del mazo que cual se ha decidido seleccionar para jugar.

Antes de pasar a ver la ejecución de una partida se mostrará la última funcionalidad dentro del menú de colecciones, seleccionando el botón ***My Cards***. Al interactuar con este botón, se enviaría un mensaje al servidor TCP correspondiente con el nombre de usuario como dato. El servidor se encargaría de realizar la acción correspondiente y buscar para este usuario el listado de cartas que le pertenecen.

Para que no se llegase a enviar un listado demasiado extenso como respuesta a esta acción, se optó por crear un método de paginación en esta ventana. Por tanto, ahora se enviaría un nuevo mensaje conteniendo el nombre del usuario del cual se desean obtener las cartas, junto con el número de página en la que se encuentra en esos momentos. De esta manera, el programa que se encarga de realizar el acceso a la ontología tan solo devolvería las cartas necesarias que se vayan a mostrar en esa página, y no todas ellas, mejorando así la rapidez de los mensajes y, por tanto, el tiempo de respuesta entre que el usuario pide ver sus cartas y cuando realmente se muestran.

Figura 4.25: Imagen del menú de cartas

En un primer momento, se mostraría la página primera, mostrando las 8 primeras cartas que se han recibido como respuesta a la petición. En esta pantalla se podría visualizar una flecha abajo a la derecha, permitiendo entonces pasar a la siguiente página. En el momento en que se pase página, aparecería otra flecha abajo a la izquierda, la cual permitiría regresar a la página anterior. En el caso de llegar a la última página, tan solo se visualizaría la flecha de regreso a la página anterior ya que, al no haber más cartas en la cuenta, no debería permitir avanzar sobre páginas vacías.

Para finalizar con la arquitectura del menú principal, se mostrará el botón de ***Options***, el cual abre el menú de opciones. Este botón es accesible tanto con un usuario con sesión iniciada como por un usuario que aún no ha iniciado sesión en el juego, ya que los ajustes deberían estar disponibles para su modificación desde cualquier instante.

Como este proyecto se centra en el sistema de combate mediante agentes y en las conexiones con servidores y el uso de una ontología, realmente no tiene una funcionalidad clara en este momento, pero se espera que en un futuro haya un amplio abanico de opciones para poder modificar tanto las opciones gráficas, como las de audio cuando se implanten.

Figura 4.26: Imagen del menú de opciones

Ya se ha visto la arquitectura detrás del menú principal, viendo cómo se vincula con gran parte de la arquitectura de la ontología, sobre todo en referente a la gestión de los usuarios y la visualización o creación de mazos y cartas del usuario en cuestión.

En el siguiente punto, se explicará la arquitectura que se esconde tras una partida del videojuego. Para ello, se deberá haber iniciado sesión y, al mismo tiempo, tener un mazo seleccionado como mazo activo con el que jugar.

4.3.3.2. Arquitectura de la partida

Este punto tratará sobre la estructura y relaciones que hay detrás de la partida del videojuego. Para ello se mostrará que scripts se esconden tras esta escena, viendo las dependencias entre estos y el pensamiento que ha habido detrás de ellos, además de su funcionamiento. Se empezará visualizando una imagen del primer momento en el que se crea una partida.

Figura 4.27: Imagen del inicio de una partida

A continuación, se describirán los elementos que se pueden visualizar en la imagen anterior. De izquierda a derecha, se encuentra primero la puntuación. A continuación, en la parte central, se observa una zona sombreada, esta se trata del tablero, donde se dispondrán las cartas que se vayan a jugar. Finalmente, en la derecha se ven varios elementos, entre ellos se pueden ver ver los mazos, situados de forma simétrica entre ellos y, en la parte central de esta zona, se observa un botón con el texto ***PLAY***.

Aunque son invisibles para el jugador, también existen dos áreas donde se van a disponer las cartas que los jugadores tienen en la mano, tanto para el jugador como para el enemigo. A continuación, se muestran dónde estarían estas áreas. [11]

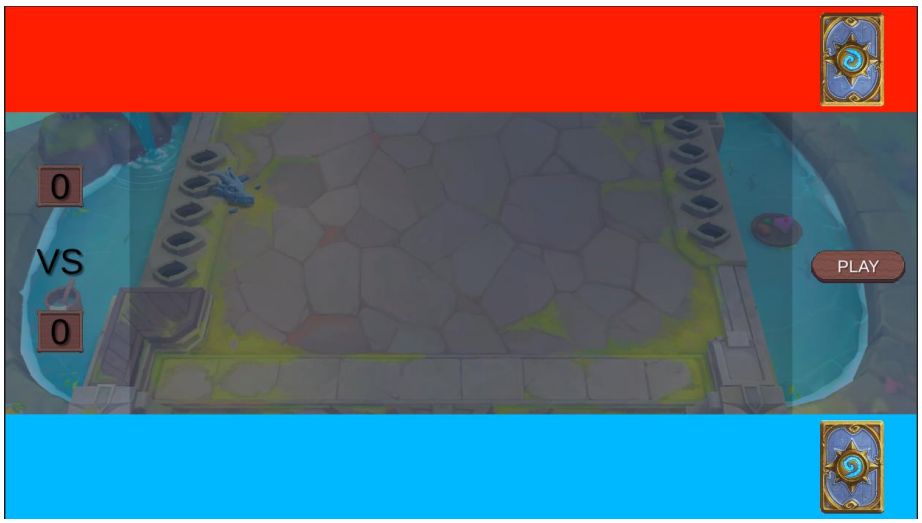

Figura 4.28: Imagen del área oculta de cada jugador

Ya se han visto los elementos que conforman este primer contacto con una partida del videojuego. A continuación, se muestra un diagrama UML para ver que se esconde tras de estos, observando los scripts que componen cada elemento en la pantalla.

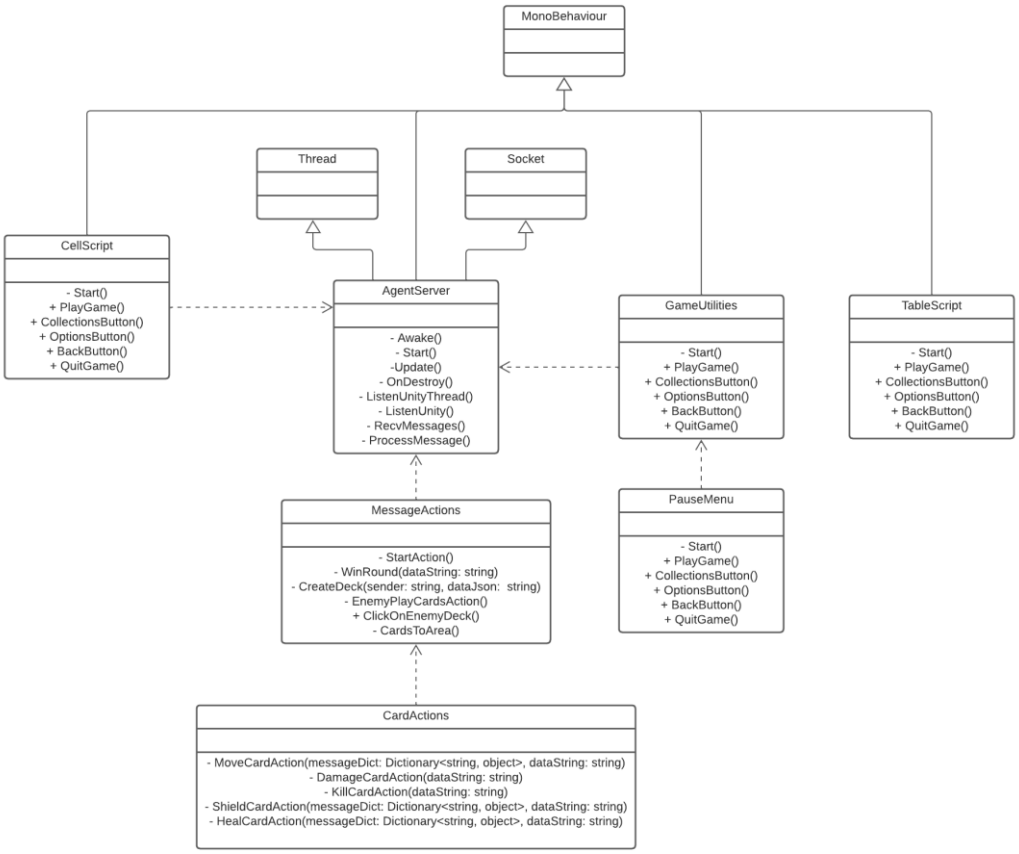

Figura 4.29: Diagrama base del inicio de una partida

Como se puede observar en el diagrama anterior, la clase ***AgentServer*** es el centro de toda la partida, así que, para poder llegar a entenderla mejor, se van a ir explicando antes las demás clases, empezando por ***TableScript***.

TableScript es un programa que se ejecuta nada más empieza la partida, este se encarga de rellenar el objecto al que hace referencia el script lleno de objetos de tipo celda. El programa ***CellScript*** sirve para realizar las acciones necesarias al disponer una carta en una celda del tablero. Por una parta, al pasar el cursor por una celda, esta comprobará si hay o no alguna carta en esta, en caso afirmativo pintará el color de la celda de un leve color verde translucido, dando a entender que la celda está disponible para disponer la carta deseada, en caso contrario, se tintará de un color rojizo, indicando así que no es posible colocar la carta en el tablero.

Fuera de la función estética que tiene este script, también cuenta con la gestión relacionada con las listas activas con las cartas disponibles tanto en el mazo, la mano, el área de

cada jugador, y las cartas que hay en juego actualmente, tanto de forma global como de forma individual, es decir, por cada jugador. Más adelante, se verá cómo se gestiona este proceso.

Ya se ha mostrado el uso del tablero y las celdas, ahora se observará el programa de ***GameUtilities***, este script ofrece las herramientas necesarias para poder interactuar con los botones de la partida, entre ellos el botón ***PLAY***, ambos mazos y el menú de pausa.

Al clicar uno de los mazos, se seleccionará la primera carta dentro del mazo del jugador deseado y se procederá a eliminarla del mazo y añadirla la mano del jugador y, por supuesto, instanciándola en el área correspondiente.

Al interactuar con el botón ***PLAY***, se dejará constancia a la clase ***AgentServer*** de que el jugador está preparado para jugar las cartas, por tanto, se enviará el mensaje correspondiente al agente Manager, pasado al próximo estado y, por ende, cambiando al turno del jugador enemigo.

Si en algún momento se quisiese salir de la partida se debería abrir el menú de pausa, esto se hace con la tecla escape y abre una nueva ventana con un botón que permite al jugador salir al menú principal. Aunque el nombre de esta función sea menú de pausa, realmente no pausa el videojuego, esto se debe al tratarse de un videojuego en tiempo real.

Figura 4.30: Menú de pausa de una partida

Para realizar las acciones necesarias a la hora de interactuar con el menú de pausa existe el script llamado ***PauseMenu***, que es el encargado de comprobar si se ha realizado una solicitud para abrir el menú o no y, encargarse de cerrar los sockets abiertos, así como de apagar los agentes en marcha.

El programa ***AgentServer*** se encarga de iniciar el subproceso necesario para inicializar el agente Manager, al mismo tiempo, se encarga de crear un socket de escucha en la dirección IP localhost junto con el puerto 8000.

Este programa cuenta con una cola de acciones, a la cual se está accediendo en todo momento para comprobar si hay alguna acción que ejecutar. Esta cola de acciones se actualiza siempre que se procesa algún mensaje desde el *sub-thread* receptor de mensajes y la acción que se deba realizar ha de cambiar alguna variable o ejecutar parte del código en el hijo principal.

Esta clase también extiende un nuevo programa, ***MessageActions***, que es el encargado de realizar todas las acciones correspondientes a la llegada de cada mensaje. Por otra parte, de esta clase se extiende ***CardActions***, la cual se encarga de procesar las acciones correspondientes a las cartas.

Ya se tiene clara la arquitectura y la apariencia inicial de una partida del videojuego, ahora queda por ver dónde y cuando entran en acción las cartas. Por eso, a continuación, se va a describir y mostrar cómo se hace. [11]

Figura 4.31: Imagen de las cartas dentro de la partida

Al iniciar una partida, se crean las instancias de cartas necesarias en el área de cada jugador, estas en un principio son 5 cartas, el máximo número de cartas que puede tener un jugador en mano en cada momento. El resto de las cartas del mazo se guarda en la lista correspondiente, esperando a ser invocada pulsando sobre el mazo.

Estas cartas son instancias de objetos de tipo ***Card***, con un programa asociado llamado ***CardScript***, el cual se encarga de dotar a la carta de sus atributos, entre otras cosas. También tiene métodos relacionados con el uso de las cartas, entre ellos mover una carta a la mesa o pasar sobre ella, que aumentaría su tamaño para poder ver mejor sus estadísticas.

Por otro lado, este script también contiene los métodos necesarios para activar las animaciones a la hora de recibir los mensajes correspondientes a la habilidad que están ejecutando sus agentes. Para realizar las animaciones, se han creado varios scripts dependiendo de la animación, entre estos se encuentran ***MeleeAttack*** y ***Shield***, que irán vinculados a sus respectivos objetos que realicen la animación.

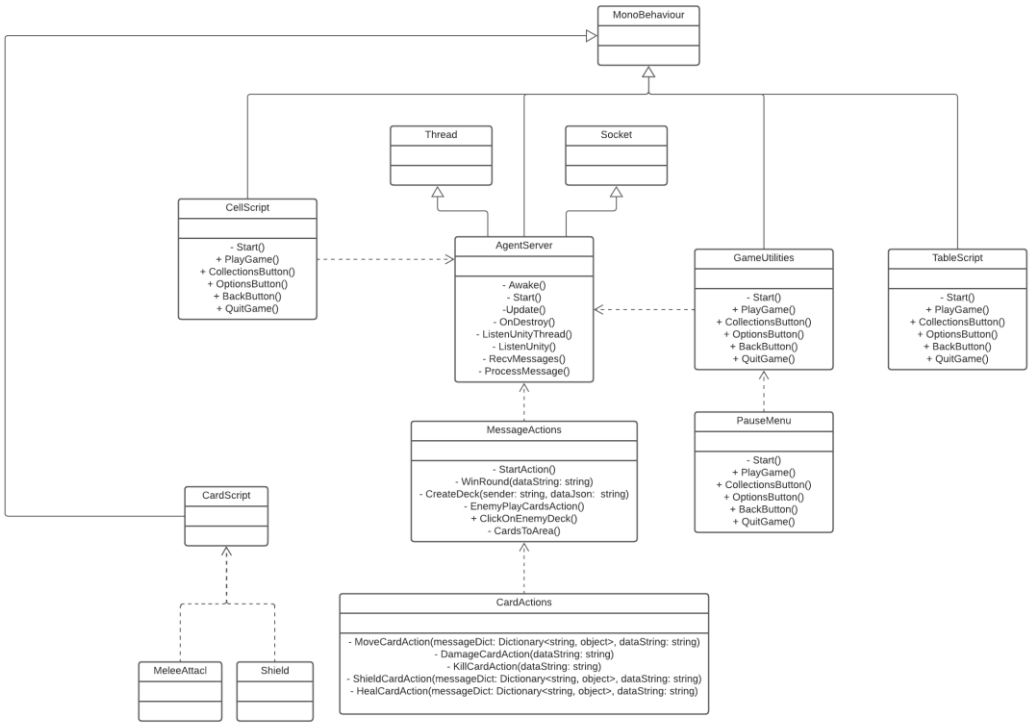

Figura 4.32: Diagrama de una partida junto con las cartas

Con lo visto hasta ahora, casi se ha finalizado de hablar sobre la estructura de este proyecto, ahora solo quedaría ver un último detalle. A la hora de jugar nuestras cartas, de forma automática el enemigo mueve cartas aleatorias al tablero y, al hacer esto, envía un mensaje al socket avisando de esta acción, por tanto, empezaría el combate.

El combate está guiado por un tiempo límite, este tiempo se ejecuta en segundo plano en un script llamado ***Timer***, el cual se encarga de actualizar el objeto con el temporizador a cada segundo, restándole esta cantidad de tiempo.

Figura 4.33: Imagen del temporizador de la partida

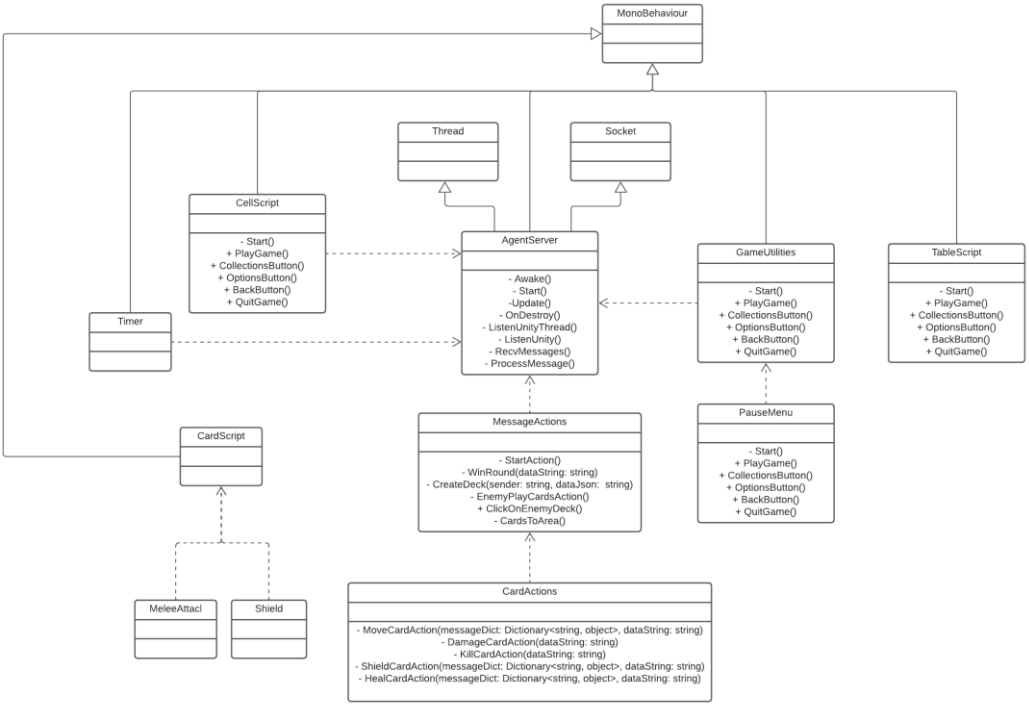

Figura 4.34: Diagrama final de una partida junto con el temporizador

En estos momentos se puede afirmar que se ha visto toda la arquitectura que se esconde detrás de una partida, desde el inicio de esta hasta conforme crece mediante el paso del tiempo, juntando la arquitectura inicial con la de las cartas y el temporizador. Al final se puede observar que es una arquitectura bastante compleja con muchos elementos relacionados entre ellos.

En el siguiente punto, se hará hincapié en como se llevaría a cabo una partida, como se interactuaría con los elementos y el orden de las acciones que hay tras estas interacciones.

4.4. Desarrollo Detallado

4.4.1. Ontología

Para poder gestionar la ontología sin tener que recurrir al entorno que otorga ***Protégé*** a cada momento, se ha decidido por crear varios scripts para acceder a ciertas funcionalidades claves de esta, entre ellas la gestión de usuarios y la creación o gestión de mazos y cartas. Ambas funcionalidades se explicarán más en detalle a continuación, mostrando diagramas UML de los tres scripts clave para el uso de la ontología con sus funciones usadas para cada caso.

OwlManager es el programa escrito en Python junto con la librería OwlReady2 encargado de la gestión de usuarios. Este se inicia en segundo plano en el momento de acceder al menú principal del videojuego y se encarga de escuchar en un socket las peticiones dadas mediante la interacción de los botones en Unity.

Para una explicación donde quede más clara las funcionalidades de este programa, se va a explicar de forma detalla los métodos imitando el proceso de un nuevo usuario del videojuego. Antes de iniciar esta explicación, se mostrará el socket que comunica este programa con el videojuego.

Este socket, al igual que los demás y de los que se hablará más adelante, está registrado como servidor TCP en la dirección IP local, "localhost" o 127.0.0.1, junto con el puerto 8002. Al iniciar este programa es cuando empieza el bucle de escucha en un hilo secundario del socket nombrado. El usuario, al interactuar con el menú del juego, se conectará como cliente al socket, enviando los mensajes con los datos necesarios para la ejecución de las acciones correspondientes y su posterior devolución al cliente como respuesta, mostrando entonces lo demandado. La información de los mensajes viene dada en formato *json*, con una forma similar a *{"action": x, "data": y, ...}*.

OwlOntology es la clase desde que realmente se accede a la ontología. Esta clase principalmente se encarga de la creación y gestión tanto de usuarios como de la colección perteneciente a cada usuario.

En cuanto a la gestión pura de usuarios, es decir, a la hora de tratar únicamente con la clase ***CUser*** de la ontología, se han creado distintos métodos que sirven para crear o acceder a individuos de este tipo en la ontología y, o bien visualizarlos para obtener cierta información, o bien para editarlos. A continuación, se visualizará el proceso de búsqueda en la ontología a la hora de hacer uso de los métodos en referencia a la gestión del usuario en un diagrama UML, desde la llegada de la petición hasta la devolución de su correspondiente respuesta.

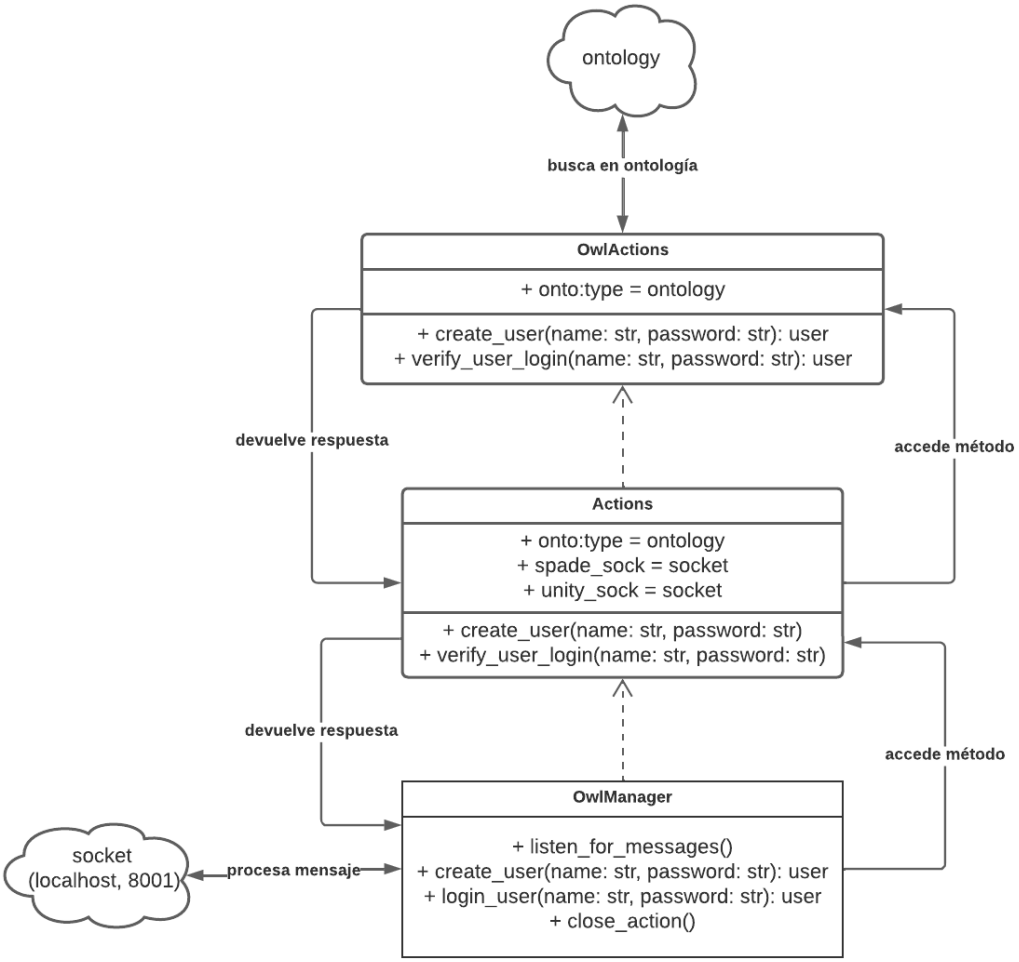

Figura 4.35: Diagrama UML del proceso de la gestión de un usuario de la ontología

Al recibir un mensaje con la acción deseada en el cuerpo a través del socket, la clase ***OwlManager*** se encargará de hacer llegar esta petición al método correspondiente de ***OwlOntology***, a través de la clase ***Actions***. ***OwlOntology*** se encargará a su vez de acceder a la ontología con la información dada y ejecutar la acción deseada, devolviendo entonces una respuesta de confirmación a ***Actions***, que a su vez acabará por llegar a ***OwlManager***.

En cuanto a la gestión de colecciones, es decir, aquella gestión relacionada con la creación o visualización de cartas y mazos se han creado una amplia selección de funciones capaces de, por una parte, acceder a la ontología para recuperar los datos necesarios y, por otra, poder transformar estos datos en tipos legibles y transferibles a través de los sockets de envío y recepción.

En esta sección, se hará hincapié en varios métodos imprescindibles para la correcta creación de cartas y mazos para lograr tener una vasta variedad de cartas y mazos diferentes y únicos entre todos los jugadores.

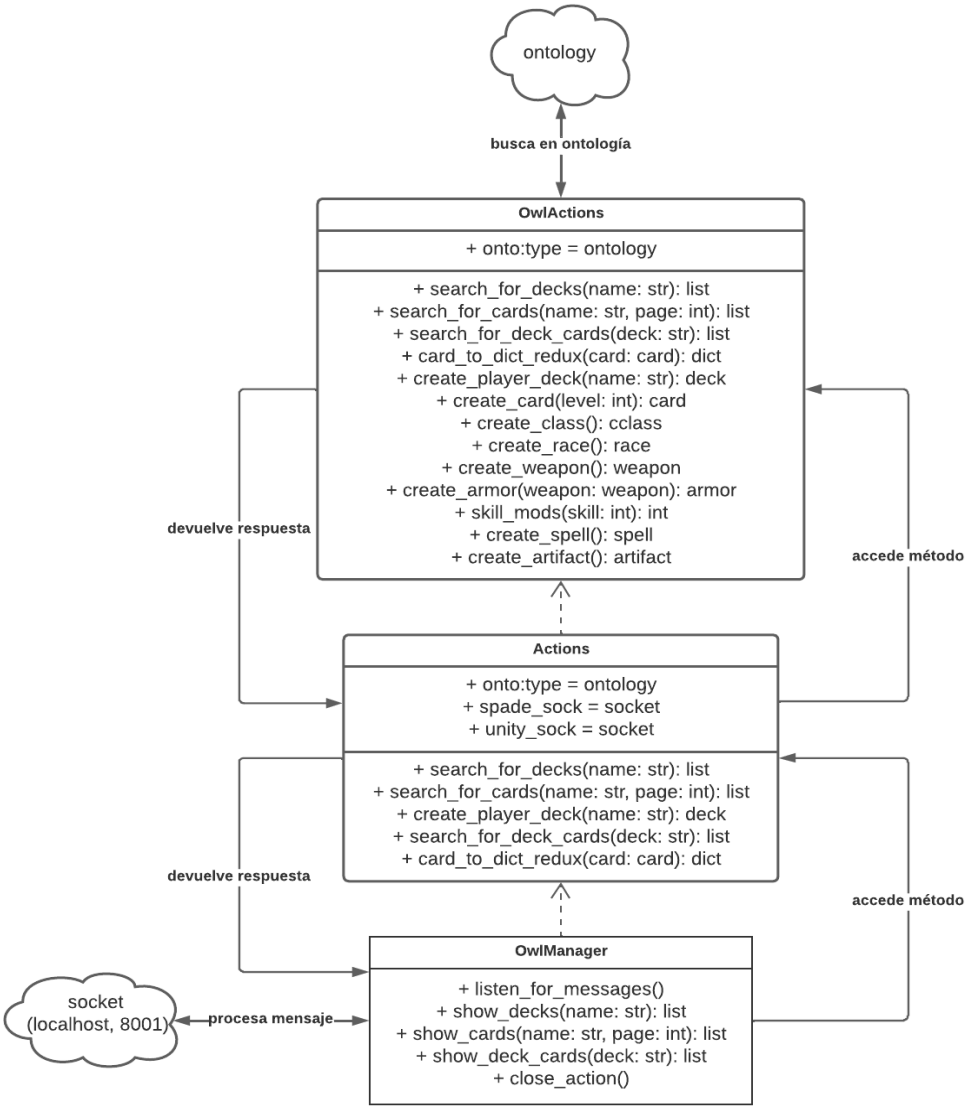

Figura 4.36: Diagrama UML del proceso de la gestión de colecciones de la ontología

Como se puede ver, la mayor parte de los métodos tiene que ver con buscar o crear mazos y cartas. Estos son muy similares, por una parte, los métodos que se encargan de la búsqueda se encargan de recorrer una a una todas las propiedades de los individuos, en este caso las cartas, y añadir sus valores a los diccionarios cuyas claves son estas propiedades. Si se tratase de un mazo, se añadirían las cartas a una lista de diccionarios.

Por otra parte, las funciones encargadas de crear mazos conllevan una larga sucesión de otras funciones, ya que para crear un mazo hay que crear una lista de cartas y, para cada carta, hay que crear sus propiedades tanto de datos como de objetos, y para esto hay distintos métodos que se encargan de crear los individuos correspondientes de forma aleatoria.

Ahora ya se entiende la mayor parte de la gestión de la ontología, ciertos puntos se verán más adelante, ya que estos son resueltos en la propia ejecución de los agentes.

4.4.2.Agentes

Los agentes se gestionan de forma muy similar a la ontología, esto se debe a que tanto agentes como ontología usan clases comunes, como son la clase ***Actions*** y la clase ***OwlActions***, las cuales se encargan de, entre otras cosas, acceder a la ontología para recoger y modificar información y, hacer las operaciones necesarias para el cálculo de las acciones de las cartas.

Ahora bien, para el correcto funcionamiento de los agentes, tanto del Manager como de cada carta que este en juego en cada momento, debe existir un orden de ejecución y, al mismo tiempo, estos deben tener un ciclo de vida. Aquí es donde entra en juego el servidor XMPP y el método por el cual se inicializa cada agente.

Para que un agente pueda ser creado y empezar su comportamiento, debe existir un servidor XMPP activo al cual conectarse. Para este proyecto, después de varias pruebas en distintos servidores, se optó por el servidor **Lightwitch.org**. Lightwitch.org es un servicio gratuito de mensajería XMPP donde se conectan los agentes. Para conectar un agente a este servidor, este debe seguir un formato al estilo *your_jid@lightwitch.org*. De esta manera, el agente ya sabrá donde ha de conectarse.

El primer paso que debe hacerse para conectar de forma correcta a los agentes es iniciar el Manager. Para esto, desde un *sub-thread* al iniciar la partida en el videojuego, se iniciará este script, iniciando así el Manager y conectándose al servidor de ***lightwitch***.

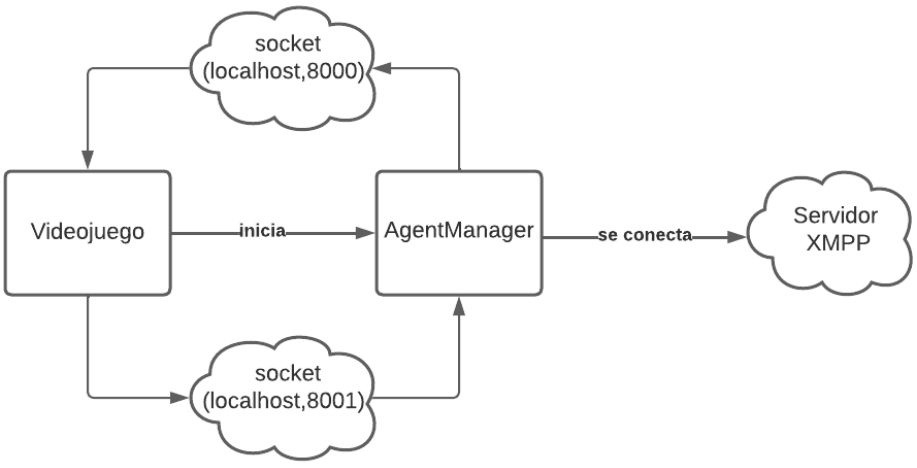

Figura 4.37: Diagrama del inicio del AgentManager

A través del videojuego, a la hora de jugar una carta y disponerla en el tablero se envía un mensaje en formato *json* con la información necesaria para crear un agente. Por ejemplo, se juega una carta de nombre card1 en la posición del tablero (0, 3). En ese instante, se enviaría el

mensaje correspondiente a través del socket conectado al puerto 8001 y, el ***AgentManager*** crearía el agente carta correspondiente con los datos necesarios.

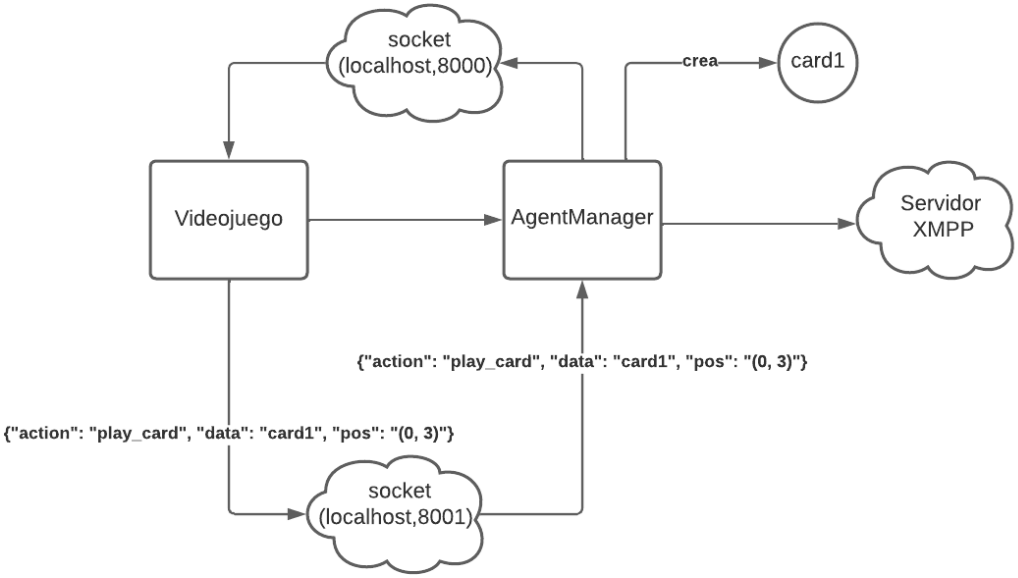

Figura 4.38: Diagrama de ejemplo de una creación de un CardAgent

De esta manera se podrían crear todos los agentes que el videojuego permitiese, el límite poniéndolo en las propias reglas del juego. Pero esto no sería todo, ahora mismo solo existen los agentes creados, pero de alguna forma hay que ponerlos en marcha y, para hacer esto, desde el videojuego se debe darle al botón ***PLAY***.

Cuando se pulse este botón se enviarán las señales necesarias para cambiar de estado en el agente y al cambiar de estado será el turno del enemigo. El enemigo de forma automática dispondrá de las cartas que vaya a jugar en el tablero y acto seguido enviará la señal necesaria al agente Manager para iniciar los agentes y así empezar a jugar.

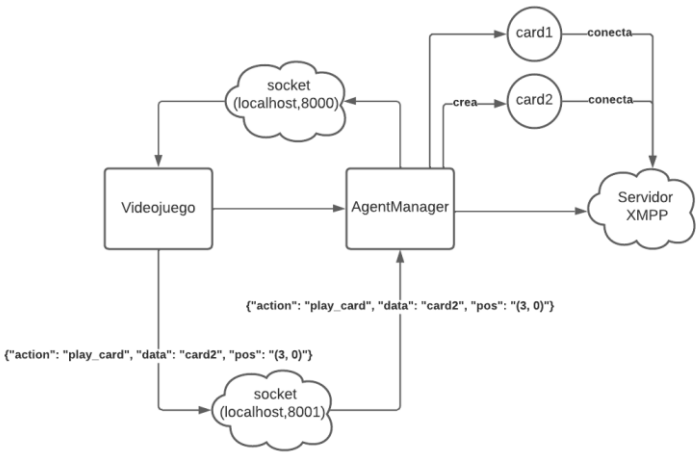

Figura 4.39: Diagrama de ejemplo del inicio de varios AgentCards en el servidor XMPP

Para finalizar, los agentes que representan a las cartas ejercen sus acciones y, cuando una de las cartas llegue a 0 de vida, es decir, muera en combate, se acabará el comportamiento de la carta y se eliminará completamente del juego.

Cuando uno de los jugadores sea elegido como ganador a través de las rondas ya no quedará ningún agente carta conectado al servidor XMPP, y el Manager tendrá que avisar al videojuego del fin de la partida y su resultado, para posteriormente acabar con su comportamiento y desconectarse del servidor XMPP.

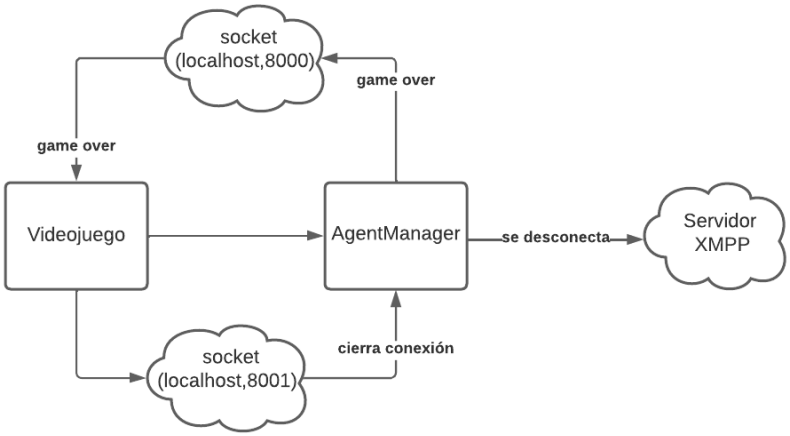

Figura 4.40: Diagrama de ejemplo del cierre del AgentManager

Ahora ya se ha visto el ciclo de vida de los agentes, así como la forma en que han de inicializarse y su orden, durante los próximos puntos se profundizará en cómo son iniciados los agentes desde dentro del videojuego y como se interactúa con estos de forma más detallada.

4.4.3.Unity

En un primer instante, al iniciar la partida desde el menú principal, se crea la nueva escena. Como paso inicial al empezar esta, la clase ***AgentServer*** se encargaría de iniciar dos *sub-threads*, donde uno de ellos iniciaría un socket de escucha con la dirección localhost y el puerto 8000. El otro hilo, iniciaría una terminal *cmd* iniciando el programa ***AgentManager*** donde este, a su vez, iniciaría otro socket de escucha también la dirección localhost, esta voz con el puerto 8001.

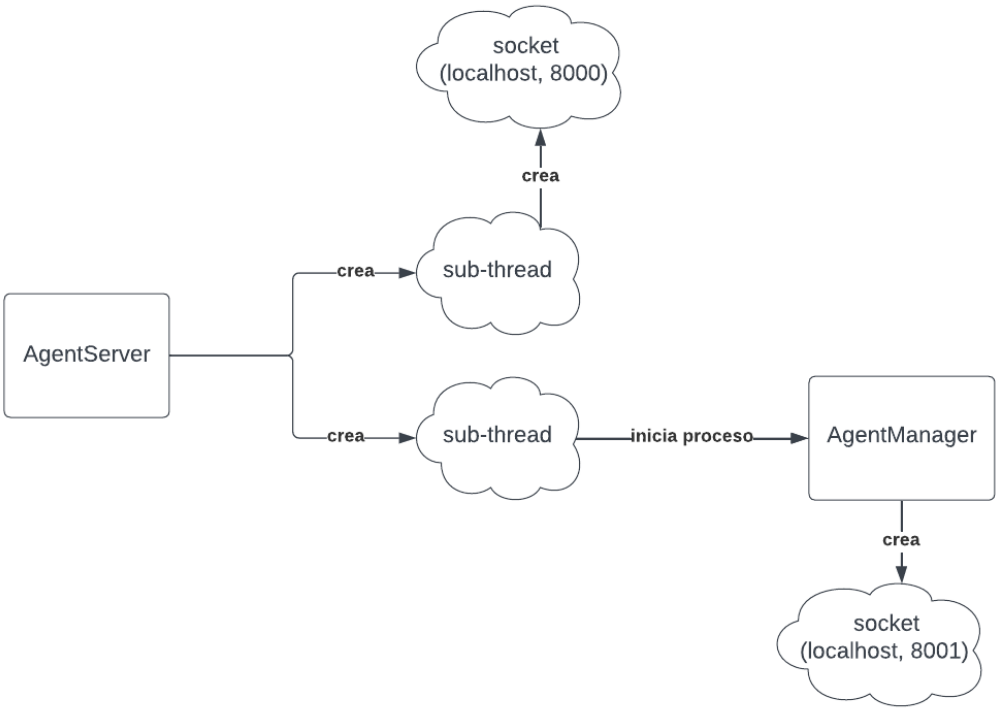

Figura 4.41: Diagrama del inicio de los sockets

En estos momentos, todos los procesos están listos para enviar y recibir mensajes. En esta etapa el Manager ya ha inicializado todas sus variables y empieza su comportamiento. Para empezar, entra en su primer estado, ***PREPARE_DECKS***, donde se envía la señal al socket del ***AgentServer*** indicando que el juego puede empezar y este, a su vez, devolviendo el mazo que se ha seleccionado en el menú de colecciones.

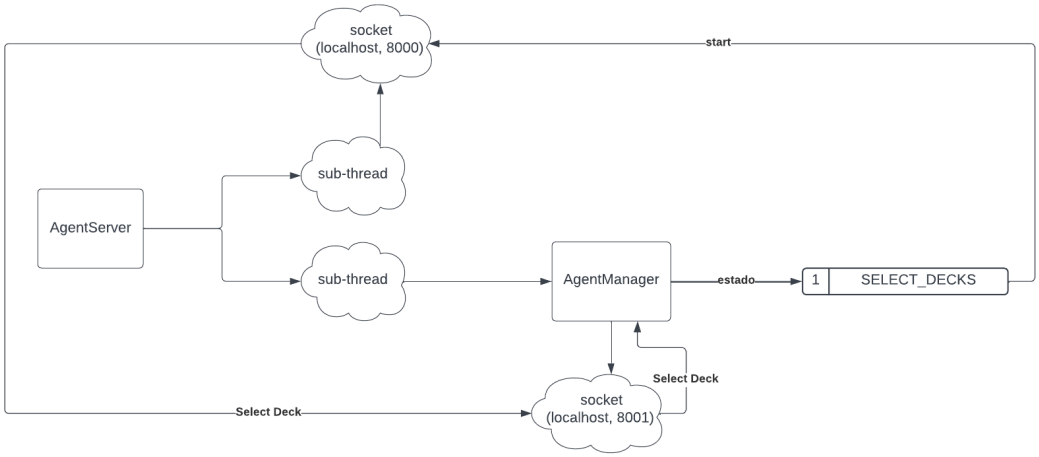

Figura 4.42: Diagrama del primer estado del Manager

En este momento, con el mazo ya seleccionado, el agente Manager pasaría al siguiente estado, ***PREPARE_DECKS***, en este estado se crea aleatoriamente el mazo del rival, y ambos mazos se enviarían en formato de listas de diccionarios al ***AgentServer***, para ser recogidos en las variables adecuadas en este script.

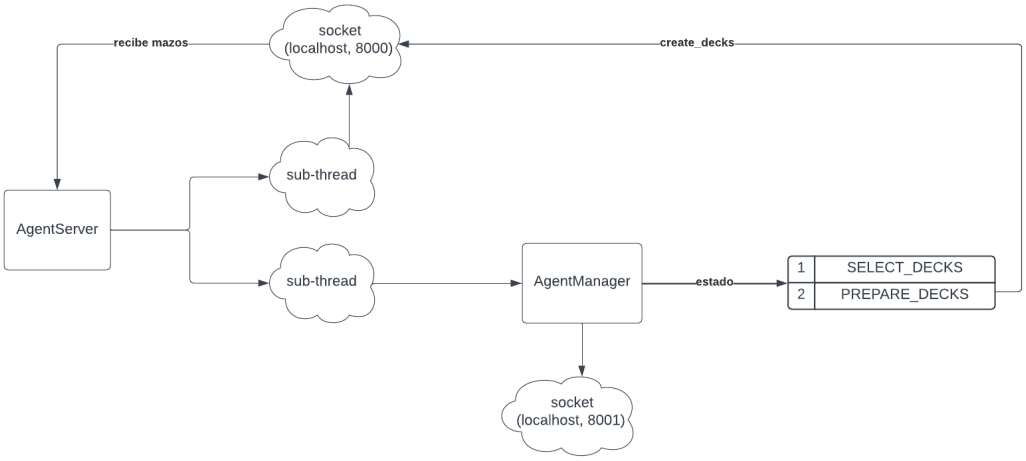

Figura 4.43: Diagrama del segundo estado del Manager

Al recibir los mazos, por una parte, se añaden las primero cinco cartas de estos a las manos de los jugadores, creando sus instancias en el área su respectivo jugador, por otra parte, el resto de mazo se guarda en las variables referentes a cada jugador.

Pasando a ver cómo se actualizarían las variables en este momento, con un ejemplo donde el mazo del jugador cuenta con las cartas numeradas de la uno a la cinco en la mano y de la once a la quince en el propio mazo, y el mazo del enemigo, es decir, la máquina, dispone de las cartas numeradas desde la seis a la diez en la mano, y de la dieciséis a la veinte en su mazo.

PlayerHand	[card1, card2, card3, card4, card5]
EnemyHand	[card6, card7, card8, card9, card10]
PlayerDeck	[card11, card12, card13, card14, card15]
EnemyDeck	[card16, card17, card18, card19, card20]

Figura 4.44: Tabla inicial de las manos y mazos de los jugadores

Inmediatamente después de enviar los mazos, el agente Manager cambia a su siguiente estado, que tan solo sirve como transición entre los estados encargados de trabajar con los mazos y los estados referentes a las rondas de la partida.

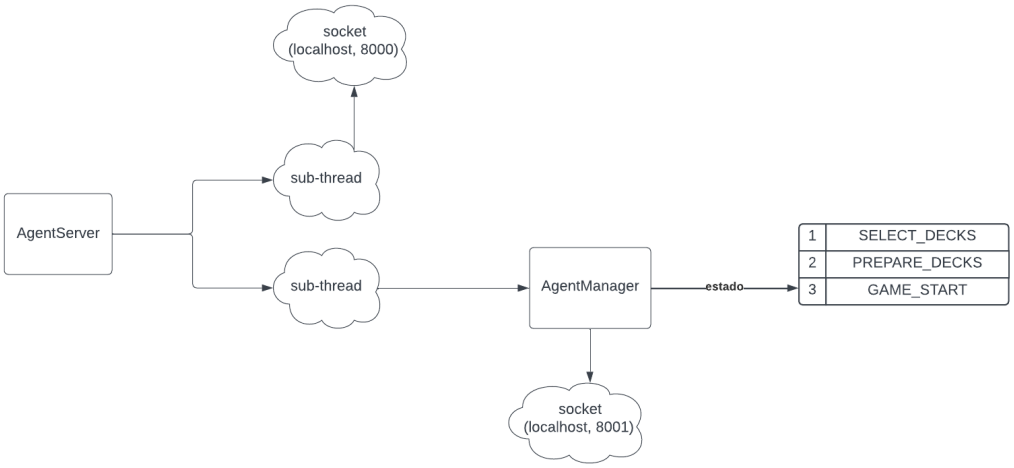

Figura 4.45: Diagrama del tercer estado del Manager

Acto seguido, hace la transición al siguiente estado, ***PLAYER_PLAY_CARDS***, el cual se encarga de enviar el mensaje adecuado al socket escuchando en localhost con puerto 8000, avisando al videojuego de que el jugador está listo para iniciar su turno, actualizando las variables que hacen referencia a la disponibilidad de jugar del jugador y, por tanto, dejandole mover sus cartas a la mesa, pero impidiendo que el rival pueda hacerlo.

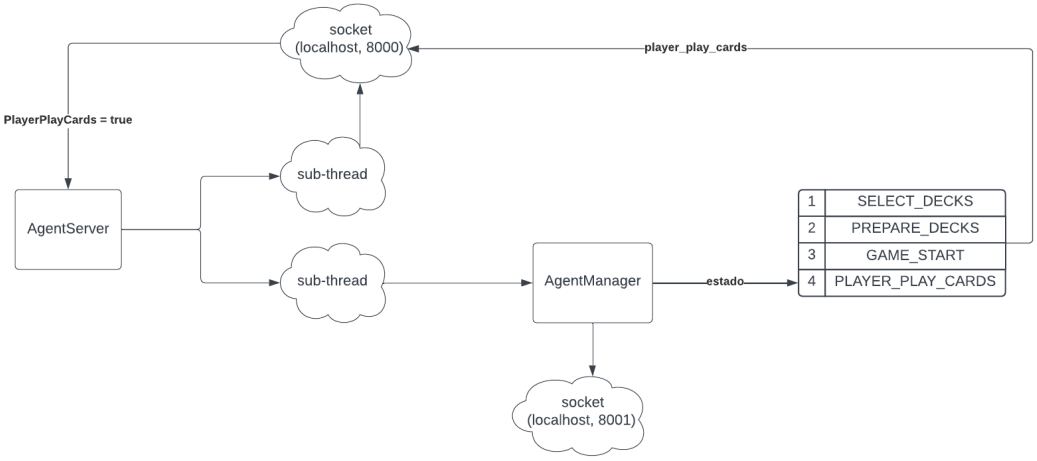

Figura 4.46: Diagrama del cuarto estado del Manager

En estos momentos el jugador tendría control absoluto para mover las cartas que quisiese al tablero, teniendo en cuenta las restricciones que existen en este. En cuanto a restricciones para jugar cartas, entendiendo jugar una carta como coger una carta de la mano y posicionarla en el tablero.

En el supuesto caso donde el jugador jugase la carta de nombre “card5”, y eligiese que va a colocarla en la celda (4, 4) arrastrándola al tablero, se mostrarían que celdas están disponibles para poder colocarlas sobre estas y cuales no, gracias al programa ***CellScript***.

Figura 4.47: Imagen del posicionamiento de una carta en el tablero

Ahora que se ha posicionado la carta donde el jugador quería, se enviaría un mensaje al socket de escucha del agente Manager de forma instantánea con el nombre de la carta y la posición. Al recibir este mensaje, el Manager se encargaría de crear el nuevo agente, buscando la carta en la ontología por su nombre y añadiéndole esta posición como atributo.

A continuación, se comprobará como quedarían las listas de cartas, ahora tanto en el propio videojuego como en las listas dentro del Manager.

PlayerHand	[card1, card2, card3, card4]
EnemyHand	[card6, card7, card8, card9, card10]
PlayerDeck	[card11, card12, card13, card14, card15]
EnemyDeck	[card16, card17, card18, card19, card20]
CardsInTable	[card5]
PlayerCardsInTable	[card5]
EnemyCardsInTable	[]

agents	[card5]
player_agents	[card5]
enemy_agents	[]

```
[[[False],[False],[False],[False],[False],[False]],
 [[False],[False],[False],[False],[False],[False]],
 [[False],[False],[False],[False],[False],[False]],
 [[False],[False],[False],[False],[False],[False]],
  [[False],[False],[False],[False],[True],[False]],
 [[False],[False],[False],[False],[False],[False]]]
```

Figura 4.48: Tablas del estado actual de las listas de cartas, agentes y el tablero

Como se puede observar, se ven en ***AgentServer*** tres listas más, donde se guardan por una parte las cartas que hay globalmente en juego y, dos listas en referencia a las cartas en juego de cada jugador. Como actualmente solo hay una carta en juego, las listas de ***CardsInTable*** y ***PlayerCardsInTable*** son idénticas.

Dentro de ***AgentManager***, se observa por una parte el tablero, donde se indica con un valor booleano, True si la celda está libre, False en caso contrario. Por otro lado, se pueden ver las tres variables de tipo lista, donde se guardan las referencias a los nuevos agentes creados, indicando cada carta. Al igual que las listas de ***AgentServer***, en estos momentos, tanto la lista global de agentes como la lista de los del jugador sería igual, al solo haber un agente en juego.

Tras jugar las cartas que desee y pueda, el jugador deberá avisar al juego haciendo clic sobre el botón de ***PLAY***, este envía un mensaje al socket del agente avisando que ya puede acabar de gestionar su perfil y puede encargarse de gestionar al enemigo.

Al recibir el mensaje, el Manager se encarga de devolver como respuesta la notificación para el videojuego referente a la supresión de nuestra capacidad para jugar y, por último, cambia de estado.

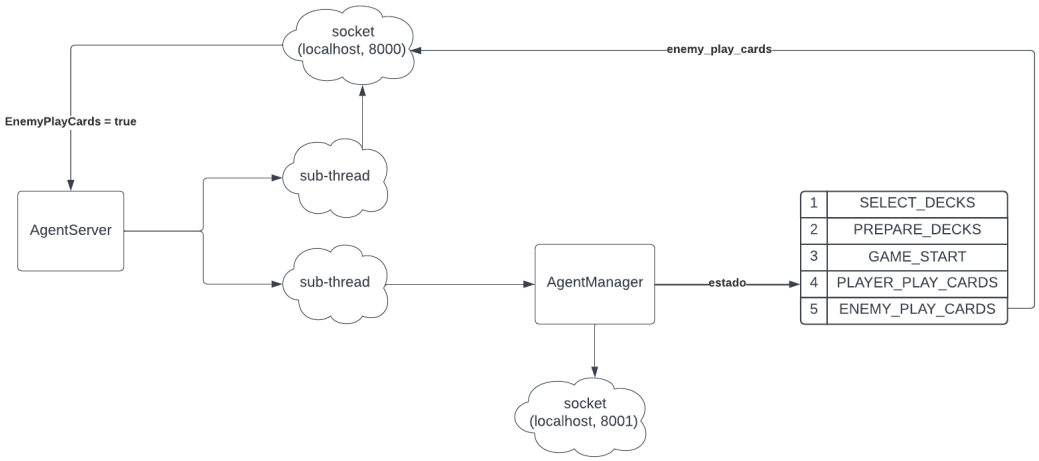

Figura 4.49: Diagrama del quinto estado del Manager

Ahora es el turno del enemigo, este al llegar su turno se encarga de escoger varias cartas aleatorias, siempre manteniendo la restricción de coste, para sacar al tablero y jugarlas. Estas las posiciona de forma totalmente aleatoria y acto seguido envía las cartas a través de un socket, junto con sus posiciones.

Figura 4.50: Imagen de las cartas del enemigo posicionadas

Como se puede observar en esta imagen, de forma análoga al estado anterior, se envían las cartas y posiciones como datos en un mensaje al socket del mánager para crear sus correspondientes agentes.

PlayerHand	[card1, card2, card3, card4]
EnemyHand	[card8, card9, card10]
PlayerDeck	[card11, card12, card13, card14, card15]
EnemyDeck	[card16, card17, card18, card19, card20]
CardsInTable	[card5, card6, card7]
PlayerCardsInTable	[card5]
EnemyCardsInTable	[card6, card7]

agents	[card5, card6, card7]
player_agents	[card5]
enemy_agents	[card6, card7]

```
[[[False],[False],[False],[False],[False],[False]],
[[False],[False],[True],[False],[False],[False]],
[[False],[False],[False],[False],[True],[False]],
[[False],[False],[False],[False],[False],[False]],
[[False],[False],[False],[False],[True],[False]],
[[False],[False],[False],[False],[False],[False]]]
```

Figura 4.51: Tablas de las listas de las cartas, los agentes y el tablero

El mayor cambio se puede distinguir en las listas correspondientes a las cartas del enemigo y las listas globales. Por una parte, al añadir las cartas enemigas al tablero se ha actualizado la lista perteneciente a esta y, al mismo tiempo, se añaden a las listas globales también estas nuevas cartas y agentes. Esto se decidió hacer así por la comodidad que otorga tener una lista con todas las cartas en juego en momentos en los que hay que iterar sobre todas estas, buscar alguna en concreto u ordenarlas de alguna forma. Al mismo tiempo se ve que la mesa también cambia modificándose los valores allá donde las cartas están posicionadas.

Nada más el enemigo juega sus cartas, se encarga de enviar una señal al agente Manager indicando que ya puede empezar el combate. En este estado, se ordena la lista global de agentes según el atributo de prioridad de las cartas. Luego, recorriendo esta lista, se inician los agentes de forma secuencial, realizando sus acciones. Antes del inicio de cualquier agente, se comprueba si se cumple la condición de victoria de algún jugador, si se diese el caso se rompería el bucle y se realizarían las acciones referentes a la puntuación.

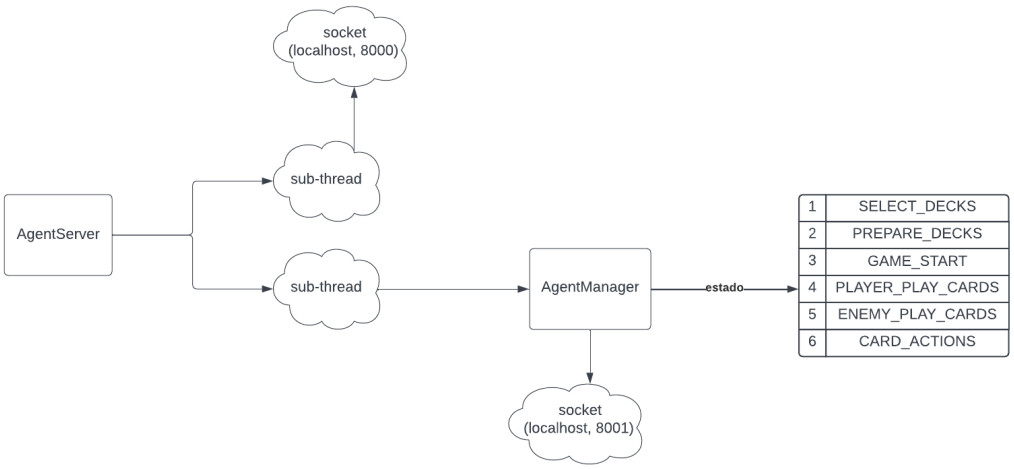

Figura 4.52: Diagrama del sexto estado del Manager

A continuación, se hará hincapié en los distintos resultados que existen al acabar una ronda, y más tarde en las acciones de las cartas. Cuando se lleva a cabo una batalla, se decide al mejor de tres. En un principio, si tras una iteración del bucle al completo no se ha decantado por la victoria de uno de los jugadores, se repetiría el mismo estado. Si en algún punto del combate uno de los jugadores se quedase sin cartas en el tablero significaría su derrota y por tanto se acabaría el bucle y se otorgaría un punto al jugador victorioso. En este punto, dependiendo de los puntos que tenga cada jugador se hace la transición al estado ***PLAYER_PLAY_CARDS***, devolviendo las cartas restantes en el tablero a la mano de cada jugador y empezando de nuevo las rondas donde se deben jugar las cartas. Si uno de los jugadores llegase a tener los puntos suficientes como para ganar la partida, se haría la transición al estado ***GAME_OVER***, encargado de finalizar con la partida, el cual se verá con más detalle más adelante.

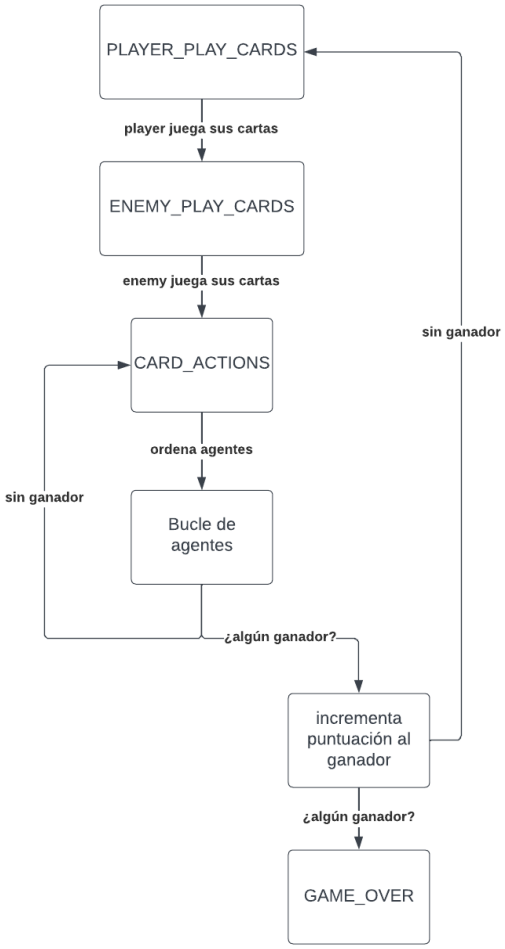

Figura 4.53: Diagrama del bucle del estado CARD_ACTIONS

En el caso de que no hubiese ganador y se repitiese el estado de ***CARD_ACTIONS*** no se realizaría ninguna acción nueva. En la situación donde hubiese un ganador de la ronda, pero no se decidiese por un ganador de la partida y se hiciese la transición el estado de ***PLAYER_PLAY_CARDS*** antes de empezar con las acciones correspondientes a la gestión de una nueva ronda, se enviarían las cartas vivas al área del jugador, que será el victorioso, y se establecerá la puntuación en los marcadores. A continuación, se verá una imagen del nuevo estado de la partida, tanto en la pantalla del videojuego como de las listas activas.

Figura 4.54: Imagen del inicio de una nueva ronda

PlayerHand	[card1, card2, card3, card4]
EnemyHand	[card6, card7, card8, card9, card10]
PlayerDeck	[card11, card12, card13, card14, card15]
EnemyDeck	[card16, card17, card18, card19, card20]
CardsInTable	[]
PlayerCardsInTable	[]
EnemyCardsInTable	[]

agents	[]
player_agents	[]
enemy_agents	[]

```
[[[False],[False],[False],[False],[False],[False]],
[[False],[False],[False],[False],[False],[False]],
[[False],[False],[False],[False],[False],[False]],
[[False],[False],[False],[False],[False],[False]],
[[False],[False],[False],[False],[False],[False]],
[[False],[False],[False],[False],[False],[False]]]
```

Figura 4.55: Listas de cartas, agentes y tablero al finalizar una ronda

En este momento, los jugadores que tuviesen menos de cinco cartas en su mano podrían interactuar con el mazo que le pertenece y añadir cartas del mazo a su mano, instanciándose esta en el área.

Figura 4.56: Imagen del estado de la partida tras pulsar el mazo

PlayerHand	[card1, card2, card3, card4, card11]
EnemyHand	[card6, card7, card8, card9, card10]
PlayerDeck	[card12, card13, card14, card15]
EnemyDeck	[card16, card17, card18, card19, card20]
CardsInTable	[]
PlayerCardsInTable	[]
EnemyCardsInTable	[]

Figura 4.57: Listas de cartas tras pulsar el mazo

También cabe recalcar que, las cartas que regresan a la mano del jugador no recuperan la vida que han perdido durante el combate, aunque para compensar esto se aumenta el valor de ciertos atributos clave de la carta, dependiendo de su clase, como podrían ser la destreza, constitución o fuerza.

Para finalizar, el último estado, ***GAME_OVER***, se encargaría de enviar el mensaje correspondiente a través del socket para avisar al videojuego del ganador y que se ha finalizado la partida. Por otra parte, también se encargaría de cerrar el socket que ha abierto el propio Manager y apagar este agente.

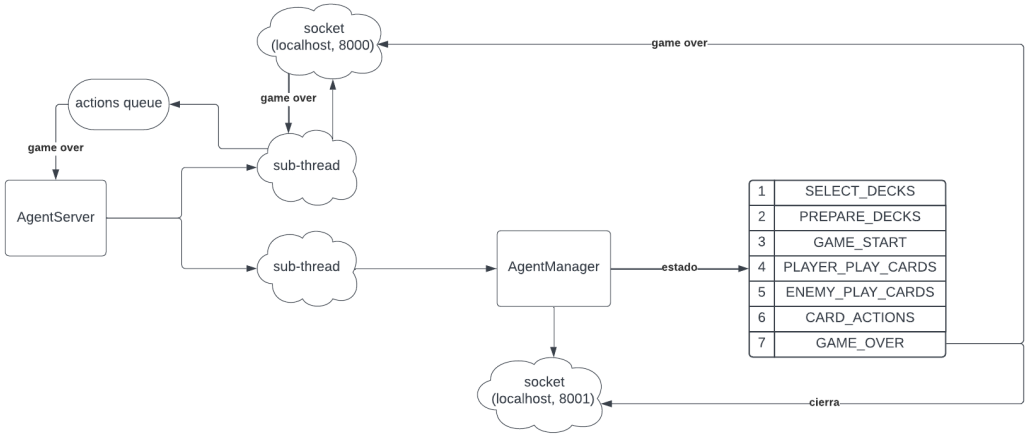

Figura 4.58: Diagrama del último estado del Manager

Al recibir el mensaje en referencia al fin de la partida, el videojuego abre un menú en pantalla indicando nuestra victoria o derrota, y añadiendo a la puntuación del usuario los valores correspondientes a esta.

Figura 4.59: Imagen del menú final de la partida

A continuación, se va a profundizar sobre las cartas y los agentes que las representan, viendo cómo son creadas y la vida de estos.

En el momento en que una carta se saca en juego para jugarla se crea su agente asociado dentro del ***AgentManager***. Este agente contiene todos los atributos que pertenecen a esta carta dentro de la ontología.

Estos agentes son creados, pero por el momento no son iniciados, para iniciarlos se debe llegar al estado ***CARD_ACTIONS*** del agente Manager, donde secuencialmente se inician los agentes siguiendo una prioridad.

Cuando un agente es iniciado en este estado, empieza su comportamiento *FSM*. En un principio entra en su primer estado, ***CARD_WAIT***, a la espera de recibir un mensaje a través del servidor XMPP y por parte del agente Manager esperando para cambiar a su próximo estado.

Inmediatamente el agente recibe este mensaje, pasando al nuevo estado, ***CARD_ACTION***, donde dependiendo de su rol y clase ejerce ciertas funciones. El procedimiento de acción de una carta siempre es el mismo. Primero, buscar la carta aliada y enemiga más cercanas. Segundo, ver si pueden ejercer su acción especial dependiendo de la clase. Tercero, atacar a la carta enemiga más cercana.

Finalmente, cuando acabe de realizar estos pasos, cambiará al próximo y último estado, ***CARD_STOP***, el cual enviará el mensaje necesario al agente Manager para indicar que el comportamiento del agente ha acabado y acabará con su ejecución.

Profundizando en ver cómo se gestiona el estado ***CARD_ACTION*** se verá cómo accede a la clase ***Actions*** para apoyarse de diversas funciones auxiliares y cómo llega a alcanzar su objetivo.

Tanto la función que calcula el enemigo más cercano, como la que calcula el aliado más próximo son idénticas. En un principio se ha de indicar el agente el cual realiza la búsqueda, como la lista de agentes con la que se calcularán las distancias. Luego, en primer lugar, se inicializa la variable de retorno con el primer agente de la lista. Más tarde, se itera sobre esta lista y, por cada iteración, se compara la distancia del agente actual con la distancia del agente cuya distancia actual sea mínima, es decir, la variable que ha de devolver el método.

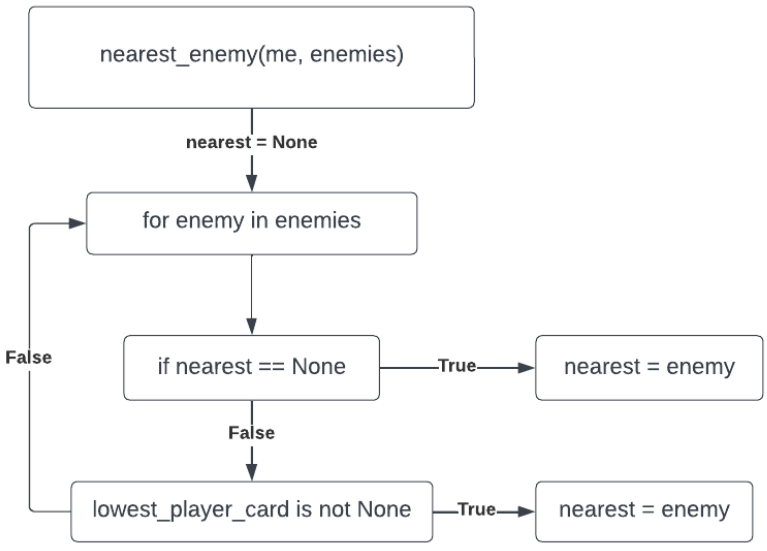

Figura 4.60: Diagrama de la función nearest_enemy

Al calcular en un principio estos agentes más cercanos, se ahorra trabajo de cálculo más adelante y sirve para cualquier acción que se pueda realizar, tanto de ataque como alguna función especial que lo requiera.

En cuanto a la habilidad especial, en un principio, para ejecutarla se debe ver el rol que ejerce y la clase a la que pertenece, ya que no todas las clases tienen ataques especiales por el momento, aunque es una ampliación que estaría bien llevar a cabo.

Por una parte, las cartas cuyo rol principal es el daño, es decir, las cartas de rol *DPS* tienen como habilidad especial hacer más cantidad de daño, por tanto, realmente la función coincide con la función propia del ataque. Las cartas que tienen un rol de mago, es decir, *MAGE*, en lugar de sumar a su daño el valor de su fuerza lo hacen con su valor de magia, por tanto, al igual que los DPS, la función es la misma que la de un simple ataque. Por último, las cartas cuyo rol es tanque, o *TANK*, no tienen ninguna habilidad dependiendo del rol, sino que cada clase dentro de este rol tiene su propia habilidad especial. Las habilidades especiales que dependen de la clase se verán a continuación.

Las clases con una habilidad especial única son Paladín, Bárbaro y Clérigo. Esto se ha elegido así, porque las clases que representaban dentro de sus roles eran muy diferentes a las demás. La habilidad especial del Paladín se trata de **escudar** a un aliado, la habilidad del Clérigo es la **curación,** y la del Bárbaro es la **Furia**. Más adelante se hará hincapié en la habilidad de escudar y curar, ahora se va a hablar de la furia. La furia es muy similar a la habilidad especial del DPS, aunque a diferencia de esta, el ataque mejorado no se da lugar cada tres ataques, sino cuando la carta de clase Bárbaro alcanza una vida inferior al cincuenta por cierto de su vida máxima, por tanto, se explicará también cuando se hable del ataque.

La habilidad de **escudar** sirve para proteger a un aliado de un ataque, esta recoge la lista de agentes aliados y calcula qué aliados tienen una vida actual igual o menor al treinta y tres por ciento de su vida máxima. Luego, elegiría entre esta lista el aliado con menor valor de vida y le otorgaría el escudo. El escudo es realmente una variable booleana que en principio está a inicializada a falso, cuando realizara esta acción una carta, la variable del objetivo pasaría a ser verdadera hasta que fuese atacada por otra criatura. Esta acción también se encarga de avisar al videojuego mediante el envío del mensaje correspondiente por el socket de Unity.

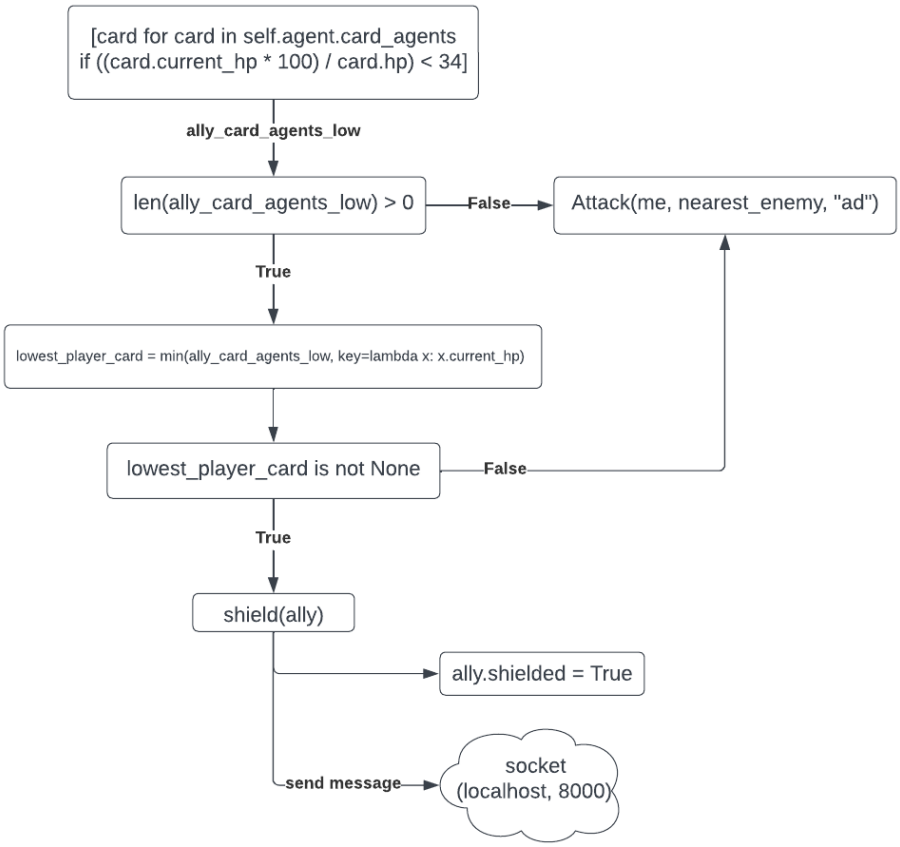

Figura 4.61: Diagrama de la habilidad especial de escudar

La habilidad de **curar** funciona de forma muy similar a la vista anteriormente. Esta se encarga de otorgar puntos de vida a la carta objetivo. En un principio, al igual que la habilidad de escudar, se buscan las cartas aliadas cuya vida sea inferior o igual al tercio de su vida máxima. Si existiese alguna, se escogería de entre estas la que tenga menor vida y se realizaría la acción, sumando a la vida de la carta objetivo el valor del atributo de nivel de la carta de clase Clérigo. Finalmente, envía un mensaje informando al videojuego de la acción realizada.

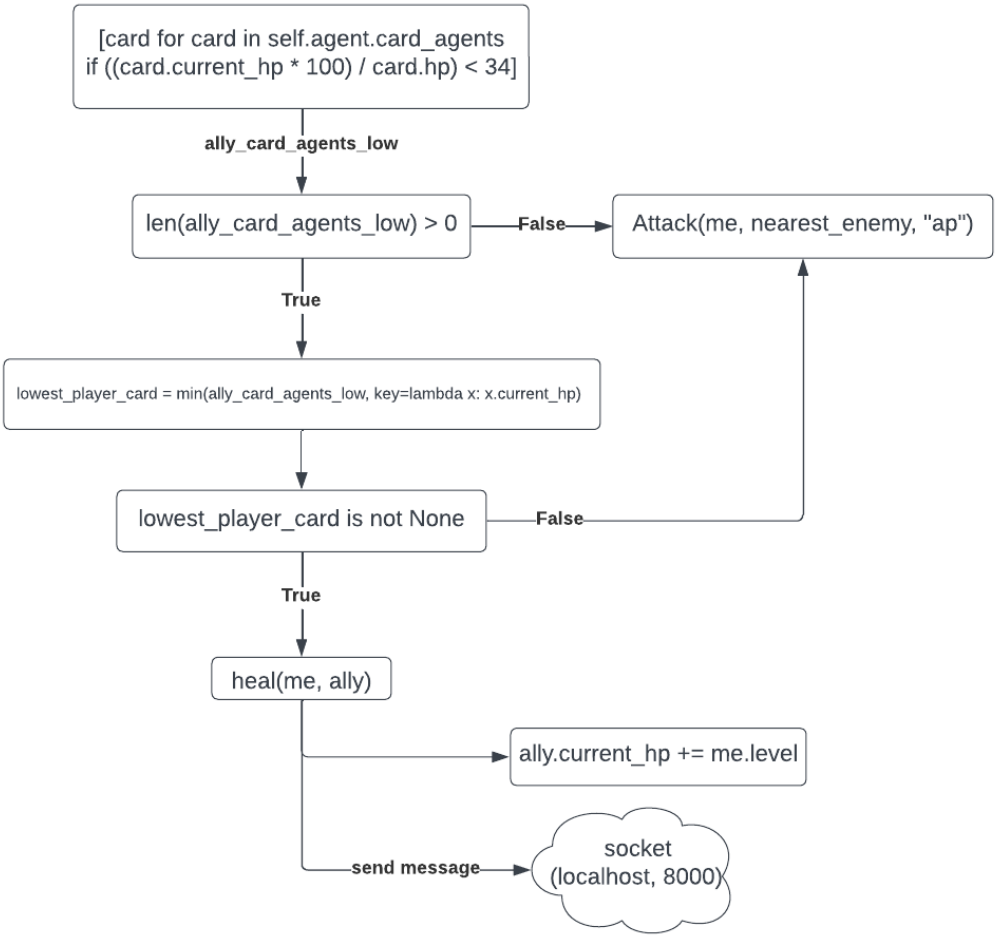

Figura 4.62: Diagrama de la habilidad especial de curar

Ya que se ha visto todas las habilidades únicas, se dirigirá nuestra atención en la acción más importante de todas, el **ataque**. La llamada a esta función tiene como parámetros el atacante, el objetivo y el tipo de daño, físico o mágico. Lo primero que hace este método es calcular el daño de forma aleatoria, esto se hace escogiendo al azar un número entre el uno y el valor del tipo de daño de la carta que ejecuta la acción. A continuación, calcula la distancia con el rival, comparándola con su atributo referente al rango. Si la carta se encontrase demasiado lejos, se moverá hacia la posición más cercana de su objetivo, siempre manteniendo el rango necesario. También se calcula si el tipo de daño es especial, como con las habilidades de los DPS o la habilidad especial del bárbaro. Como última comprobación, se chequea si el objetivo tiene un escudo activo en estos momentos o no. Si fuese así, se cambiaría el estado de la variable en referencia al escudo de la tarjeta objetivo a falso, y se enviaría un mensaje al videojuego indicando que la carta enemiga ya no está escudada. Si no es así, se le restaría el daño a infligir sumado a el atributo correspondiente a la clase de la carta atacante al objetivo. Si se diese el caso que la nueva vida de la carta objetivo fuese igual o menor a cero, se pararía el comportamiento del enemigo y se eliminaría de las listas de los agentes activos que hiciese falta, además se enviaría un mensaje al videojuego indicando que esa carta ya no está en juego. Si este

no fuese el caso, se enviaría un mensaje al socket creado por Unity indicando el daño que debe recibir la carta objetivo.

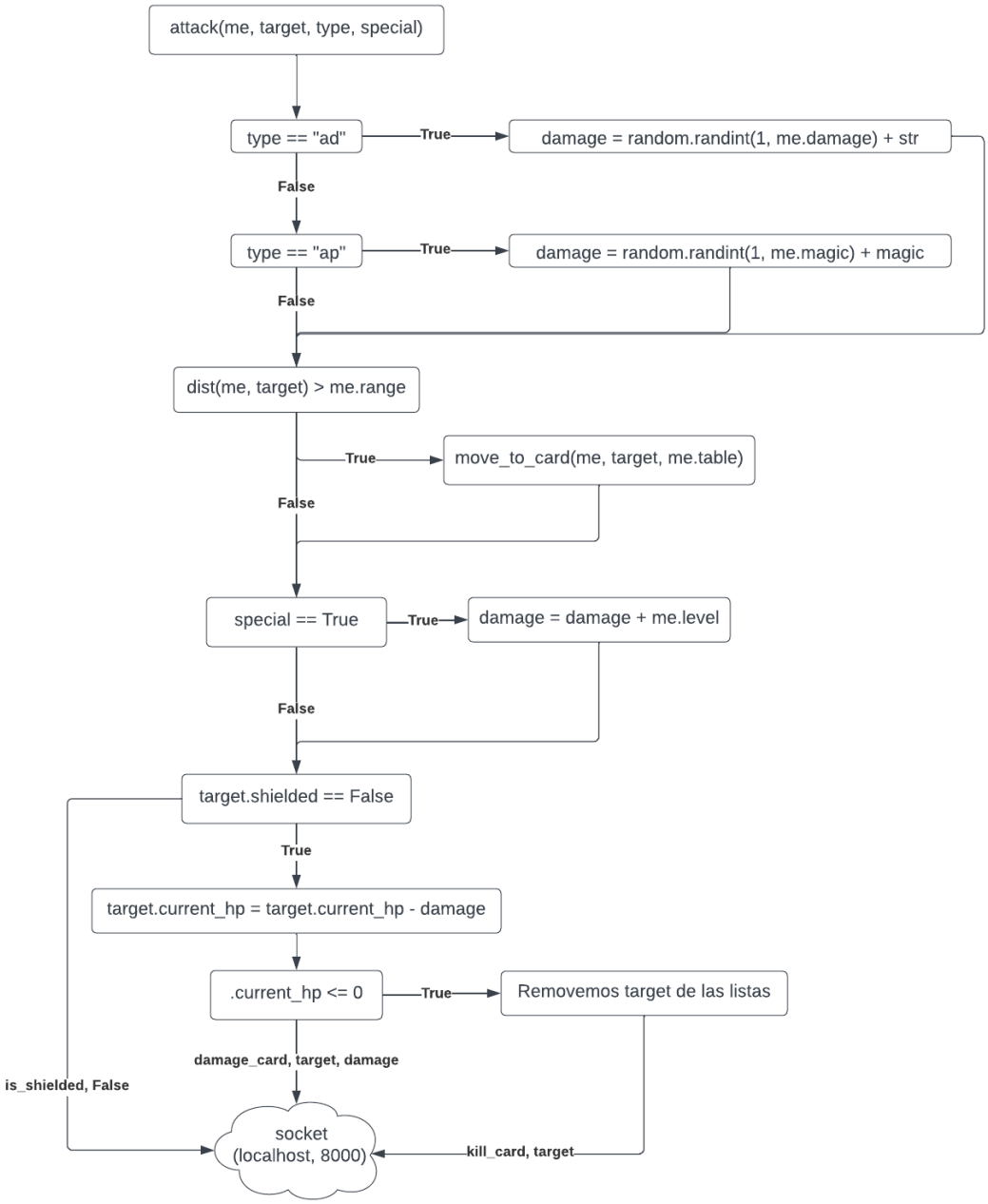

Figura 4.63: Diagrama de la habilidad de atacar

Ahora ya se ha llegado a comprender la gestión existente tras un simple ataque en este videojuego de cartas, pero también es interesante conocer qué se esconde detrás del movimiento de cada carta. El método ***move_to_card*** es la función que se ha creado para mover una carta de

su posición actual en dirección a su carta objetivo, y esta está respaldada por muchas otras funciones auxiliares que se verán a continuación.

En primera instancia, cabría comentar que las posiciones de las cartas vienen dadas en formato de cadena, y no como un tipo tupla, por eso, como primer objetivo se tiene que conseguir transformar este texto en una tupla. Luego, se observa si la carta que ha de moverse pertenece al jugador o al enemigo, esto influye de tal manera en que, cuando avanza una carta del jugador hacia una carta enemiga, esta carta ha de priorizar posicionarse en una celda que este más cerca de su área de juego que de la enemiga, y viceversa. A continuación, se comprueba si la distancia entre la carta y el enemigo es mayor que el rango de la carta y si la carta puede moverse a la posición deseada. Si fuese así, se movería las casillas necesarias para poder cumplir con las restricciones de rango de la carta. Si no fuese así, se buscaría entre otras casillas también alrededor del objetivo hasta posicionarse en alguna celda. Finalmente, se enviaría un mensaje al videojuego indicando el movimiento que ha de realizar la carta.

A continuación, se describirán algunas de las funciones auxiliares más interesantes. La función ***dist*** toma las posiciones de ambas cartas a las cuales se van a medir su posición, esta calcula la distancia de Manhattan, es decir, calcula la suma de las diferencias absolutas entre las coordenadas de cada agente, distancia = |x2 - x1| + |y2 - y1|. El método ***can_move***, recoge el estado actual del tablero y la nueva posición, por una parte, calcula si esta posición se encuentra dentro del rango del tablero, y luego comprueba que dicha posición este libre o no, devolviendo True o False. El método ***move*** se encarga de mover una carta de una posición a otra, estableciéndole a esta carta la nueva posición y modificando los valores necesarios de la variable table, en referencia a la mesa.

Antes de finalizar con esta sección se va a explicar otro tipo especial de cartas. Las cartas de tipo **Artefacto** y las de tipo **Hechizo**. Estas cartas se caracterizan por ser lanzadas sobre otra carta aliada o enemiga, por tanto, tendrán un comportamiento diferente. En un principio, estas cartas nunca son instanciadas, cuando se deja caer una carta sobre el objeto de otra carta en el videojuego se envía un mensaje al socket del Manager de la misma manera en la que se envían las cartas de tipo **Criatura**, con su nombre y su posición, con el añadido del tipo de carta de la que se trata, si artefacto o hechizo. Cuando el socket recibe este mensaje, lo procesa buscando en primer momento la carta de tipo criatura que se encuentra en la posición dada y, a esta carta, se le aplica el efecto resultante de la carta especial en cuestión. Para ello, se busca en la ontología el tipo de carta de la que se trata y, según su tipo, se le aplican unos efectos u otros a la carta objetivo.

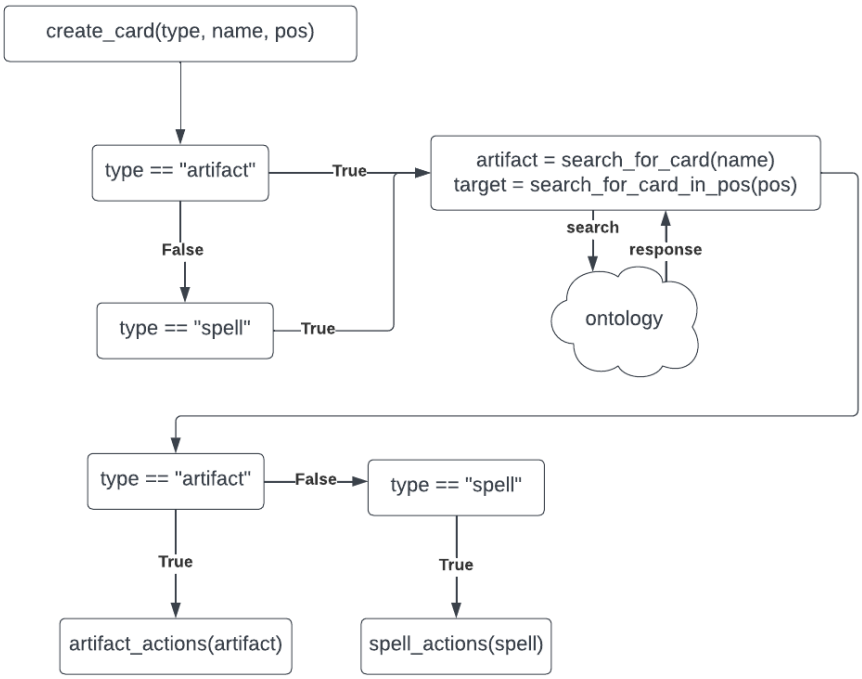

Figura 4.64: Diagrama de la creación de cartas

Ya se ha acabado con la explicación de la gestión tras el uso de cartas y los agentes que le son vinculados a estas, por tanto, se puede dar por acabado este apartado. En este momento, se debería entender no tan solo la tecnología que hay detrás del videojuego, sino también su funcionamiento y las razones por las que se ha hecho de esta manera.

5. Implantación

En este apartado se tratará sobre cómo se ha implantado el videojuego. Para ello, se ha hecho uso de una función muy útil que viene por defecto en el motor gráfico que se ha usado, Unity.

Unity contiene una herramienta de creación de *Build* automatizada, estas *Builds* se crean en la nube listas para ser ejecutadas con opción multiplataforma (con las plataformas disponibles según las que se hayan escogido previamente para este proyecto).

De esta forma, se puede acceder en cualquier momento a esta funcionalidad y, preparando las escenas como se desee, crear una *Build* lista para ser jugada.

6. Pruebas

En cuanto a las pruebas, estas se realizaban a medida que se iba avanzando el proyecto. A continuación, se explicará cómo y cuándo se iban realizando las pruebas de las tres distintas tecnologías de este trabajo.

Primero se dedicará un punto a la ontología. En esta se iban creando las nuevas clases y atributos y, en un código aparte del proyecto final, se hacían las pruebas de creación de individuos. Durante esta etapa de pruebas en la ontología se produjeron muchos cambios y se cometieron muchos errores debido a la novedad de trabajar con esta tecnología y librería, pero finalmente se consiguió tener una versión estable de la ontología, que es la que se usa hoy en día en el videojuego.

A continuación, se hará hincapié en los agentes. Para hacer pruebas con los agentes se ha hecho uso de los ejemplos básicos que hay en la documentación de **SPADE**, cada vez aumentando la complejidad de los estados del comportamiento. Cuando se consiguieron los estados necesarios y funcionales, tanto del Manager como de las cartas, solo quedaba conectar los sockets en los agentes, que ya se verá más adelante.

En **Unity**, las pruebas se hicieron de una manera mucho menos complicada. Esto ha sido posible gracias al propio editor de escenas e inspector que trae como herramientas. Gracias a estos, se ha podido observar en todo momento el transcurso de la partida, corrigiendo posibles fallos o imperfecciones.

Como último, se va a hablar de los sockets, la tecnología que ha dado más problemas. En cuanto a estos, se empezó creando pequeños programas con funcionalidades simples como un *ping*, o un *eco*. Luego, se añadía complejidad, como el paso y respuesta de mensajes simples, como texto plano o enteros. Para finalizar con las pruebas, se hicieron pruebas pasando listas y diccionarios. Para hacer esto, se tuvieron que utilizar las librerías adecuadas dentro de cada proyecto para poder transformarlas a cadenas de texto en formato *json*, y luego al recibir los mensajes, convertirlos de nuevo en el tipo de lista o diccionario correcto, para poder tratar con ellos.

Con todas las pruebas hechas, solo quedaba juntar las piezas del proyecto y trabajar con todas las tecnologías como si fuesen una sola.

7. Conclusiones

Para finalizar, hay que comentar que, a pesar de las dificultades de trabajar con varias tecnologías nuevas al mismo tiempo y conseguir que se puedan comunicar entre ellas de manera eficiente, se puede afirmar que se ha acabado por conseguir un videojuego funcional y jugable.

Se han conseguido lograr los objetivos propuestos en un principio, e incluso mejorarlos y aportar nuevas funcionalidades, como son el sistema de usuarios y el sistema de colecciones. En primera instancia no se esperaba tener que lidiar con una gestión de usuarios de la manera en que se hace, ni tener un sistema de colecciones como el que se tiene actualmente.

Con este proyecto se ha conseguido crear un juego de estrategia de combates de un jugador contra la máquina muy entretenido, con cada partida diferente a la anterior gracias a la creación de cartas aleatorias al inicio de cada partida. Además de este modo de juego, se tiene el centro del proyecto terminado, siendo un sistema de combate sólido y ampliable en cualquier momento, que se puede exportar a nuevos modos de juegos.

Hablando de la ampliación a otros modos de juego diferentes, este sistema de combate se podría exportar al proyecto de Enric Puigcerver, ya que a este le falta el sistema de combate que no se pudo realizar en su día.

7.1. Relación del trabajo desarrollado con los estudios cursados

Este proyecto cuenta con muchas tecnologías vistas a lo largo de la carrera. Entre ellas, destacan las siguientes asignaturas:

- **Programación**: Al tratarse el proyecto de un sistema informático, no puede faltar la programación, más específicamente la programación orientada a objetos o POO. La mayor parte del cuerpo de este trabajo está escrito y relacionado mediante clases y objetos, al igual que como se ha visto durante todos estos cursos en el grado.
- **Agentes Inteligentes**: El tema central del proyecto se basa en agentes inteligentes. Para programarlos se ha usado **SPADE**, una librería de **Python** para tratar con sistemas multi-agentes, desarrollada en la misma UPV.
- **Redes**: Para relacionar el proyecto de Unity junto con los agentes y la ontología se ha hecho uso de sockets, una herramienta muy útil y que otorga mucho potencial, vista en las asignaturas de redes.
- **Interfaces Gráficas**: En Unity se ha tenido que desarrollar una interfaz gráfica con la cual el usuario interactúe para poder comunicarse con los agentes y ontología, además de para poder visualizar la partida, ya que se ha usado Unity como un render de las acciones de los agentes.
- **Gestión de proyectos**: Durante todo el proyecto se ha llevado a cabo una metodología ágil, quedando semana tras semana con el tutor del trabajo y debatiendo las futuras tareas, a la par que corrigiendo el trabajo hecho.

- **Concurrencia**: Para el proyecto, sobre todo a la hora de comunicación entre los sockets y la ejecución de los agentes, se ha hecho uso de métodos concurrentes para no detener el transcurso de la partida. Además, se han usado varios hilos para arrancar los procesos encargados de iniciar los agentes y acceder a la ontología.

7.2. Reflexiones

En este último punto de las conclusiones voy a dar un punto de vista personal, hablando y debatiendo sobre cómo me he sentido a lo largo del desarrollo del proyecto.

Por una parte, me vi abrumado al darme cuenta del trabajo que supondría usar tantas tecnologías diferentes y conectarlas entre ellas. Cuanto más entraba en estas nuevas tecnologías pensé si realmente podría conseguirlo, pero fue gracias a la gente que me apoyaba, a mi compañero Enric Puigcerver y a mi tutor, Carlos Carrascosa, que me dieron el apoyo para continuar con este proyecto.

Gracias a empezar con este trabajo he conseguido adquirir nuevos conocimientos y mejorar los ya conocidos. He mejorado muchas flaquezas a la hora de trabajar, tanto en equipo, como individualmente bajo presión.

Por último, quiero comentar que he disfrutado mucho haciendo este proyecto, ya que he trabajado haciendo un tipo de videojuego que me apasiona y, al verlo como un reto que superar y completarlo, me siento realizado y satisfecho conmigo mismo.

8. Trabajos futuros

Este proyecto trata sobre la creación de un sistema de combate, exportable a muchos otros modos de juego. En un principio, viene con un modo de juego de estrategia de una batalla del jugador usuario contra la máquina, pero puede ser ampliable. Entre las ampliaciones disponibles, destacan:

- **Modo de juego online**: Creación de un nuevo modo de juego, idéntico al ya existente, pero en lugar de combatir contra la máquina programar algún tipo de método de creación de partidas para poder invitar amigos con los que jugar. Más adelante también se podría añadir un menú de búsqueda de partidas públicas para combatir con otros jugadores. Para esto se necesitaría tener un servidor.
- **Ampliar el sistema de colecciones**: En un futuro se podría crear una ampliación al sistema de colecciones, permitiendo al usuario modificar sus mazos y conseguir nuevas cartas, por ejemplo, al obtener una victoria.
- **Mejorar el apartado artístico del videojuego**: Como este proyecto se ha centrado en el desarrollo de un sistema de combate mediante agentes y no en el apartado gráfico del videojuego, se podría contratar un artista para realizar los cambios necesarios para tener una interfaz única, original y bonita.
- **Añadir nuevos tableros para el modo de combate**: Una de las posibles ampliaciones podría ser dentro de un combate tener disponible distintos tableros, como por ejemplo alguno con celdas bloqueadas o uno de mayor o menor tamaño. Esto conseguiría una mayor variedad dentro del juego y haría cada partida aún más individual de lo que ya es.
- **Integrar el sistema de combate en el proyecto de Enric Puigcerver**: Al ser este un proyecto en colaboración con Enric Puigcerver faltaría integrar este sistema de combate y su respectivo modo a su proyecto, juntando ambas tecnologías, completando así el proyecto colaborativo.

9. Agradecimientos

Como último punto, me gustaría agradecer a toda la gente que ha estado a mi lado durante este duro proyecto. Me gustaría en primer lugar a las personas que me han ayudado a realizar este trabajo, por una parte, a mi tutor Carlos Carrascosa, el cual cada semana me ayudaba con mis dudas y asistía en mi proyecto, recomendándome mejoras y dándome su opinión sobre el avance de este. También, a mi compañero Enric Puigcerver, con quien construí la primera idea del trabajo y el cual me apoyó y ayudó en todo momento.

Quiero también agradecer a mi familia, amigos y compañeros, por ser un pilar para mí durante estos años de carrera, los cuales me motivaban a continuar y me han apoyado en todo momento.

Y, por último, agradecer a todos vosotros, lectores, por mostrar interés en mi trabajo y dedicarle el tiempo a algo que he hecho con tanto cariño.

10. Referencias

[1] Owl. https://www.w3.org/OWL/

[2] MCGUINNESS, Deborah L., et al. OWL web ontology language overview. *W3C recommendation*, 2004, vol. 10, no 10, p. 2004.

[3] OwlReady2. https://owlready2.readthedocs.io/

[4] Lamy JB. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. Artificial Intelligence In Medicine 2017;80:11-28.

[5] SPADE. https://spade-mas.readthedocs.io/

[6] Palanca, J., Terrasa, A., Julian, V., & Carrascosa, C. (2020). Spade 3: Supporting the new generation of multi-agent systems. *IEEE Access*, *8*, 182537-182549.

[7] Unity. https://unity.com/es/solutions.

[8] ANDRADE, António. Game engines: a survey. *EAI Endorsed Transactions on Serious Games*, 2015, vol. 2, no 6.

[9] C#. https://learn.microsoft.com/en-us/dotnet/csharp/

[10] HEJLSBERG, Anders, et al. *The C# programming language*. Pearson Education, 2008.

[11] How to Create a Multiplayer Card Game in Unity. https://www.youtube.com/watch?v=QiKSVpyXRKI

[12] How to Create a 2D Card Game in Unity. https://www.youtube.com/watch?v=0-dUB52eEMk

[13] START MENU in Unity. https://www.youtube.com/watch?v=zc8ac_qUXQY

[14] Python. https://www.python.org/

[15] CHALLENGER-PÉREZ, Ivet; DÍAZ-RICARDO, Yanet; BECERRA-GARCÍA, Roberto Antonio. El lenguaje de programación Python. *Ciencias Holguín*, 2014, vol. 20, no 2, p. 1-13.

[16] El equipo de 3DJuegos. (2019, marzo 08) *¡Estrategas! Estos son los 10 mejores juegos de cartas*.
https://www.3djuegos.com/juegos/hearthstone-heroes-of-warcraft/noticias/estrategas-estos-son-los-10-mejores-juegos-de-cartas-190308-1407

[17] Alex, C. D. (2019, julio 1). *Qué es el Auto Chess y por qué se está convirtiendo en el nuevo género de moda*. Vidaextra.com; Vida Extra.
https://www.vidaextra.com/estrategia/que-auto-chess-que-se-esta-convirtiendo-nuevo-genero-moda

[18] XMPP. https://xmpp.org/

[19] SAINT-ANDRE, Peter; SMITH, Kevin; TRONÇON, Remko. *XMPP: the definitive guide*. " O'Reilly Media, Inc.", 2009.

[20] https://soulofkiran.itch.io/pixel-art-wooden-gui-v1

[21] https://cafedraw.itch.io/fantasy-card-assets

[22] https://ravenmore.itch.io/fantasy-icon-pack

[23] Página oficial de ***Dungeons & Dragons***. https://dnd.wizards.com/es

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**			X	
ODS 2. **Hambre cero.**			X	
ODS 3. **Salud y bienestar.**			X	
ODS 4. **Educación de calidad.**			X	
ODS 5. **Igualdad de género.**			X	
ODS 6. **Agua limpia y saneamiento.**			X	
ODS 7. **Energía asequible y no contaminante.**			X	
ODS 8. **Trabajo decente y crecimiento económico.**			X	
ODS 9. **Industria, innovación e infraestructuras.**			X	
ODS 10. **Reducción de las desigualdades**.			X	
ODS 11. **Ciudades y comunidades sostenibles**.			X	
ODS 12. **Producción y consumo responsables.**			X	
ODS 13. **Acción por el clima.**			X	
ODS 14. **Vida submarina.**			X	
ODS 15. **Vida de ecosistemas terrestres.**			X	
ODS 16. **Paz, justicia e instituciones sólidas.**			X	
ODS 17. **Alianzas para lograr objetivos.**			X	

ETS Enginyeria Informàtica
Camí de Vera, s/n. 46022. València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

El 25 de septiembre de 2015, los líderes mundiales adoptaron un conjunto de objetivos globales para erradicar la pobreza, proteger el planeta y asegurar la prosperidad para todos como parte de una nueva agenda de desarrollo sostenible. Cada objetivo tiene metas específicas que deben alcanzarse en los próximos 15 años.

Este proyecto no cumple con ningún objetivo de los ODS, aunque esto no significa que no puede llegar a vincularse con ninguno. Los videojuegos, en general, pueden llegar a relacionarse con los objetivos de desarrollo sostenible, ya que algunos de estos pueden tratarse de videojuegos educativos o tratar sobre las metas que se pretenden alcanzar con los ODS.