Argentum Online
Documentacion Tecnica

Taller de programacion I [75.42]
Primer cuatrimestre de 2020

Taiel Colavecchia - 102510 - tcolavecchia@fi.uba.ar
Franco Daniel Schischlo - 100615 - fschischlo@fi.uba.ar
Nicolás Aguerre - 102145 - naguerre@fi.uba.ar

Índice

1. Requerimientos de Software

1.1. Sistema operativo

Para compilar y ejecutar el proyecto, se debe utilizar alguna distribución de GNU/Linux de 64 bits. Se recomienda utilizar en particular alguna distribución basada en Debian.

1.2. Herramientas basicas

Tanto para compilar el cliente como el servidor, es necesario instalar GCC, Make (ambas incluidas en el paquete build-essential) y CMake. Cada una de estas puede instalarse mediante:

```
$ sudo apt-get install build-essential
$ sudo snap install cmake --classic
```

1.3. Bibliotecas

1.3.1. Cliente

Para compilar/desarrollar el cliente, se deben instalar las versiones de desarrollo de SDL. Esto puede ser logrado mediante

```
$ sudo apt-get update
$ sudo apt-get install libsdl2-dev libsdl2-image-dev libsdl2-ttf-dev libsdl2-mixer-dev
```

1.4. Servidor

El servidor no necesita ninguna dependencia adicional para ser compilado.

1.5. Para la creacion y edicion de mapas

Para crear o editar mapas, es necesario instalar el software Tiled.

```
$ sudo snap install tiled
```

2. Compilacion del cliente

Para compilar el cliente, se debe crear dentro del directorio *client* un nuevo directorio llamado *build*, correr cmake y luego correr make install:

```
~/argentum/client$ mkdir build
~/argentum/client$ cd build
~/argentum/client/build$ cmake ..
~/argentum/client/build$ make install
```

Una vez hecho esto, ya se puede correr el cliente mediante

```
~/argentum/client/build$ ./aoclient
```

3. Compilacion y ejecucion del servidor

Para compilar el servidor, el proceso es identico al de compilar un cliente: Se debe crear una carpeta build, ejecutar cmake y luego make install:

```
~/argentum/server mkdir build
~/argentum/server cd build
~/argentum/server/build$ cmake ..
~/argentum/server/build$ make install
```

Ahora, el servidor se puede ejecutar mediante:

```
~/argentum/client/build$ ./aoserver
Dispatcher: starting..
ClientDropHandler: starting..
ChangeMapHandler: starting..
ClientInitializeHandler: starting..
CommandHandler: starting..
CreationHandler: starting..
MapManager: created map: "Paseo Colon"
MapManager: created map: "Reserva Puerto Madero"
MapManager: created map: "Facultad de Ingenieria"
ServerManager: creating session for map id: 2
ServerManager: creating session for map id: 1
ServerManager: creating session for map id: 0
GameServer: SERVER STARTED.
```

Para finalizar su ejecución, simplemente se debe ingresar `'q'` en la consola:

```
GameServer: SERVER STARTED.
[...]
q
GameServer: CLOSING...
ClientsMonitor
Dispatcher: Stopped handlers
Gameloop finished
Observer finished
Observer finished
Observer finished
Sessions: finished all
Accepter: joined
Dispatcher: Joined handlers
Dispatcher: joined
MapChanger: joined
GameServer: SERVER CLOSED.
```

4. Empaquetado del cliente

Se provee un script de Makefile en el directorio `package generation` del repositorio, mediante el cual se genera un paquete de Debian que contiene el cliente listo para ser desplegado. Este paquete se genera con los assets tal y como se encuentran en `client/assets`, los indices en `client/ind`, y el ejecutable correspondiente a la ultima compilacion, tomado de `client/build/aoclient`.

```
~/argentum/package_generation$ make
```

4.1. Despliegue del cliente

Cuando se instala el cliente mediante el paquete de Debian, el binario correspondiente al mismo queda instalado en `/usr/local/bin`, los assets quedan instalados en `/usr/local/share`, y el archivo de configuracion se genera en `/argentum/` en caso de no encontrarse uno. La razon para que el archivo de configuracion no se instale junto con el paquete, y se genere por el programa en el directorio `home`, es que esto permite que luego el archivo de configuracion sea modificado sin permisos de root. Luego de ejecutar por primera vez el programa, el archivo de configuracion queda ya guardado en `/home/<usuario>/argentum/config.json`.

5. Descripción general

El proyecto es en esencia una aplicación distribuida, compuesta por dos programas: Un cliente y un servidor. A rasgos generales, el cliente no es mas que una "ventana"que muestra y envía eventos al estado del juego en cada momento, donde dicho estado se encuentra en realidad en el servidor, donde también se actualiza.

Como se trata de un juego multijugador, el servidor es capaz de atender varios clientes de forma concurrente, mientras que la conexión desde el cliente hacia el servidor es única (es decir, el cliente solo realiza *una* conexión con el servidor).

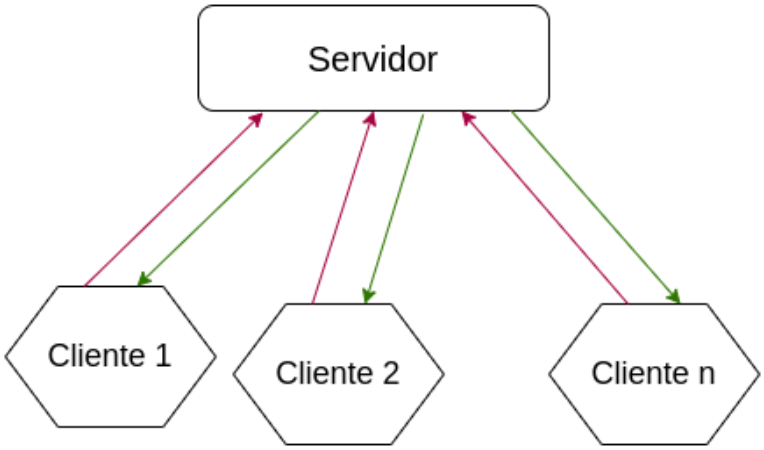

Figura 1: Diagrama general de la aplicación.

Como se mencionó, el cliente no es mas que una ventana al estado del juego que en realidad se encuentra en el servidor. Ninguna de la lógica del juego es calculada en el cliente. Los eventos del usuario (como por ejemplo aquellos que se desencadenan al presionar una tecla) son encapsulados por el cliente en una forma conocida por el servidor, y enviados a través de la red, para que sea el servidor quién decida como debería cambiar el estado del juego de acuerdo a la acción realizada por dicho jugador.

La comunicación entre el servidor y los clientes está dada por mensajes de eventos que se describirá más detalladamente en la sección de Eventos.

Desde el punto de vista del cliente, no siempre es posible enviarle al servidor las mismas acciones. Por ejemplo, a un jugador que aún no se conectó al juego con el nombre correspondiente a su personaje no le es posible indicarle al servidor que se desea mover a la izquierda (porque, después de todo, no tiene un personaje asignado). Esto nos lleva al concepto de *vistas* en el cliente, que se corresponden con las diferentes configuraciones de "ventanas"que ve el jugador mientras tiene abierto el cliente. Existen tres vistas diferentes en el cliente: La vista de inicio de sesión, que es capaz de enviar el mensaje correspondiente a la conexión inicial con un nombre de personaje; La vista de creación de personaje, capaz de enviar una solicitud de creación de personaje con los atributos deseados; y por último, la vista principal de juego, a partir de la cual se pueden enviar mensajes de movimiento, ataque, chat, etc. Cada una de estas vistas, así como también los mensajes, serán explicados con mas detalle en la próxima sección.

El servidor consta principalmente de tres partes, una encargada de mantener *actualizado* el estado del juego (la llamaremos *"GameManager"*), una segunda encargada de atender las acciones que se quieren realizar sobre el estado del juego (*"Dispatcher" de eventos*) y una última que envía las actualizaciones del juego a los clientes correspondientes (*.ºbserver" del mapa*). De las primeras

dos el servidor cuenta con una única, mientras la de la última hay una por cada mapa del juego. Al ejecutarse el servidor inicia el "GameManager"que a su vez crea todos los mapas del juego, luego de esto el servidor asocia cada mapa del juego a una "Sesion"(será la parte que conecta los mapas lógicos del juego con la comunicación con los clientes) y en cada una de estas un .ºbserver"del mapa. Una vez inicializado el servidor está listo para aceptar conexiones de clientes nuevos. Al cerrarse el servidor desconecta a los clientes de forma ordenada, finaliza el "GameManagerz cierra todas las sesiones.

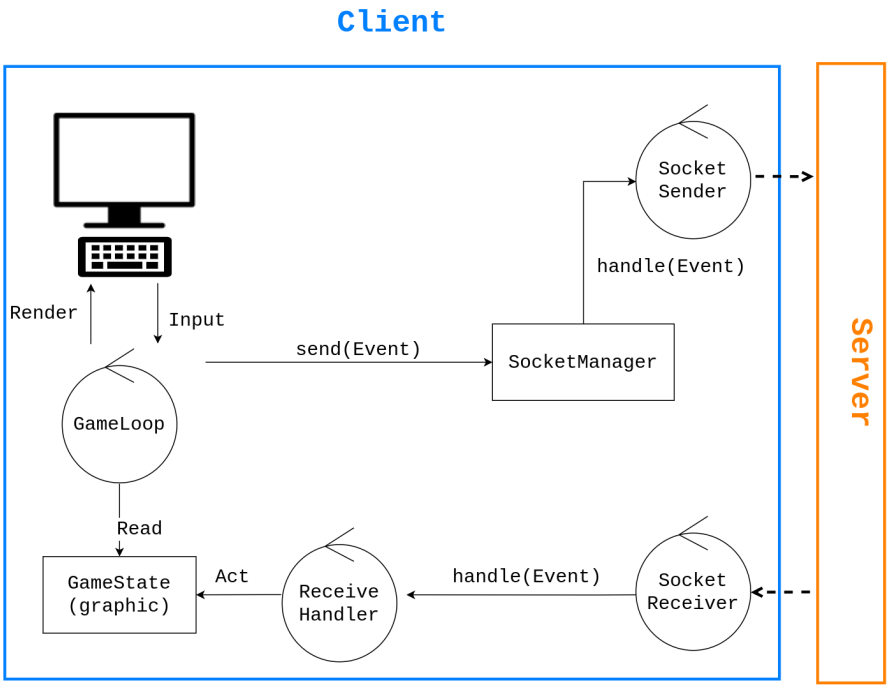

Figura 2: Diagrama la arquitectura de hilos del cliente.

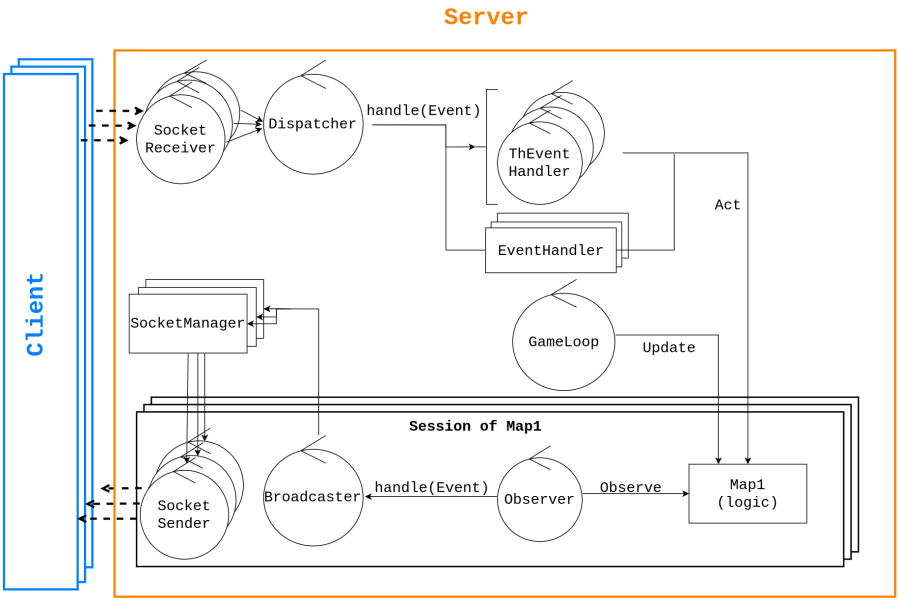

Figura 3: Diagrama la arquitectura de hilos del servidor.

6. Cliente

6.1. Primitivas de SDL

Todas las gráficas del cliente están construidas en base a un conjunto de primitivas de SDL, que no son mas que wrappers de las estructuras de SDL hechas para C, pero adaptadas para ser clases RAII de C++. Una referencia detallada de estas clases (y de todas las del cliente) puede ser hallada en la referencia del cliente.

La clase clave de este conjunto es la clase SDLSprite, a partir de la cual se hacen todas las animaciones del juego.

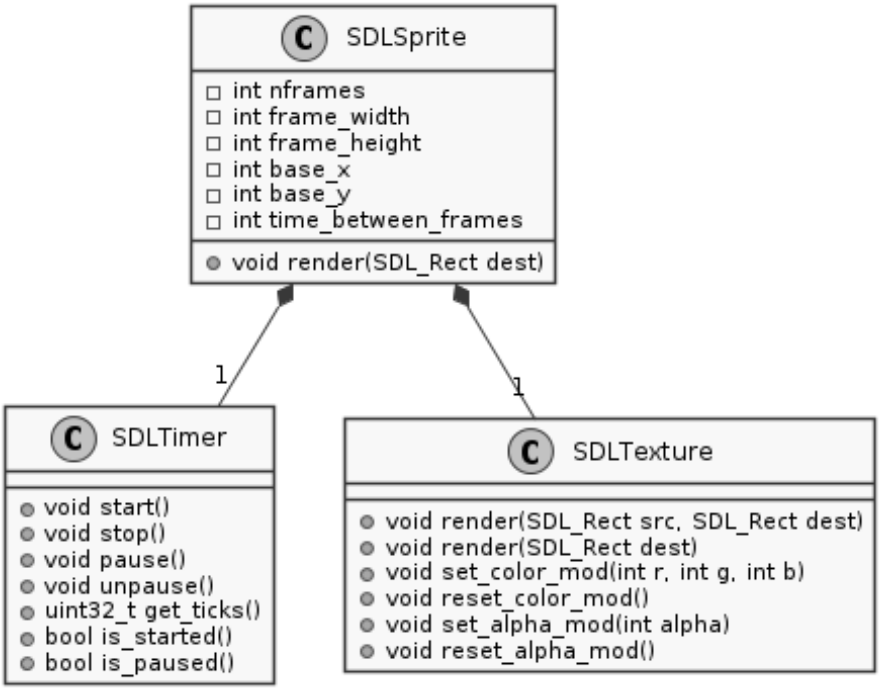

Figura 4: Diagrama de clase de SDLSprite.

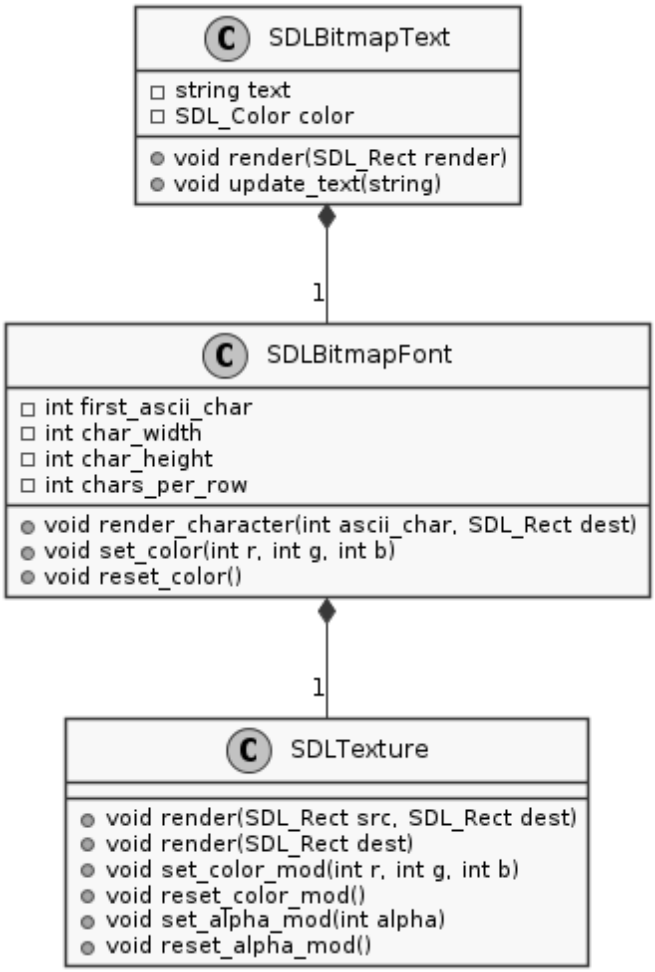

Figura 5: Diagrama de clase de BitmapText.

6.2. Engine

Una capa de abstracción por encima de las primitivas de SDL, se encuentra el *engine*. El *engine* construye, a partir de estas primitivas, un conjunto de clases que representan objetos con distintas características dentro del juego, como ser los conjuntos de sprites que representan el movimiento de un personaje en todas las direcciones, las imágenes estáticas que componen el mapa, los elementos de la interfaz de usuario, etc. Todas las unidades en el engine están expresadas en términos de las unidades arbitrarias del juego (tiles). Cada uno de estos elementos será explicado con mas detalle en esta seccion.

6.2.1. Camara y objetos Renderizables

El engine establece que todo objeto renderizable dentro de la vista principal del juego debe heredar de la clase RenderizableObject.

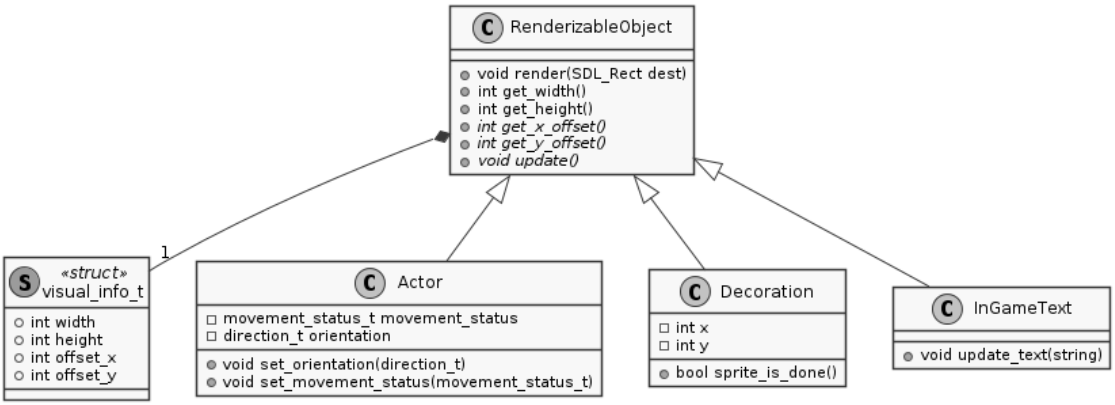

Figura 6: Arbol de herencia de RenderizableObjects.

Los dos objetos renderizables estándar son los Actors ,las Decorations y el InGameText.

Los Actors se utilizan para renderizar aquellos sprites que representen las cuatro posibles orientaciones de un mismo objeto (y se utiliza para los personajes).

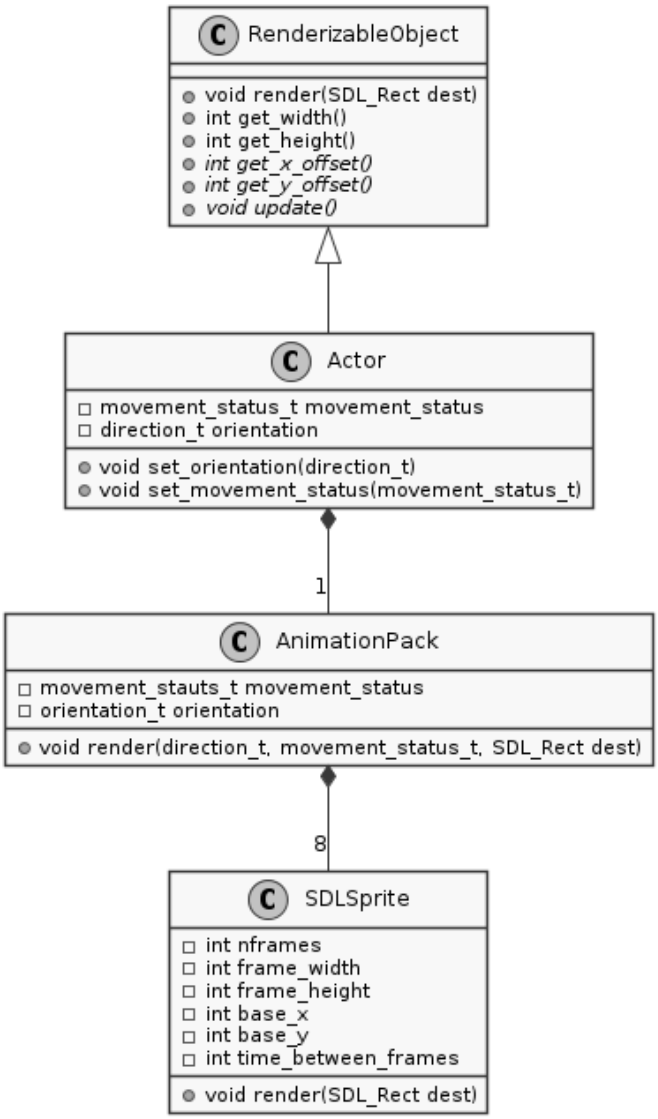

Figura 7: Diagrama de Actor.

Las Decorations se utilizan para renderizar cosas estaticas (que no se desplazan por el mapa, y por ende no necesitan tener informacion sobre todas las direcciones), lo que no implica que no puedan ser animadas. Una Decoration tiene dentro de si un sprite, que puede o no ser animado (segun la cantidad de cuadros que tenga).

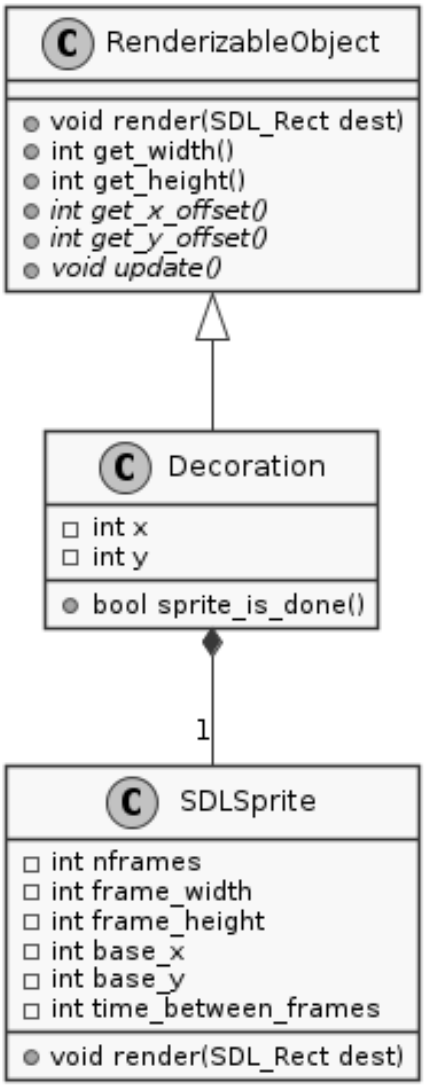

Figura 8: Diagrama de Decoration.

Finalmente, los InGameText se utilizan para renderizar nombres (tanto de jugadores como de NPCs), y para mostrar daño recibido.

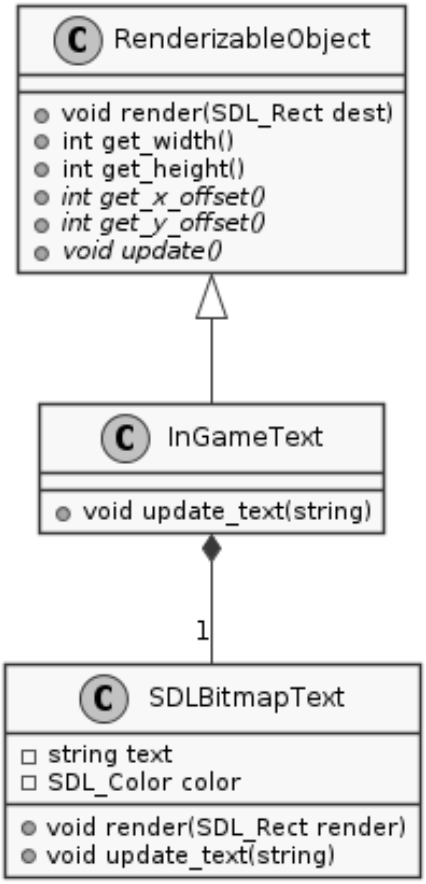

Figura 9: Diagrama de InGameText

Como se mencionó antes, ningun objeto renderizable conoce verdaderamente su posición ni su tamaño en pixeles, sino que se trabaja con unidades arbitrarias del juego. Así, las posiciones de las Decorations están expresadas en *tiles*, y los *offsets* de todos los objetos renderizables estan expresados con una cierta granularidad en terminos de tiles (el estandar es centesimas de tile, pero esto puede cambiarse en la configuracion del engine). Este offset se utiliza para desplazar la posición de renderización de los objetos respecto del origen del tile (esquina superior izquierda), y es útil a la hora diferentes objetos que comparten una posición (como el cuerpo y la cabeza de un personaje).

Cualquier objeto que herede de RenderizableObject puede ser renderizado por la Camara, que es el objeto encargado de traducir las unidades arbitrarias de juego en posiciones absolutas en pixeles dentro de la ventana para renderizar.

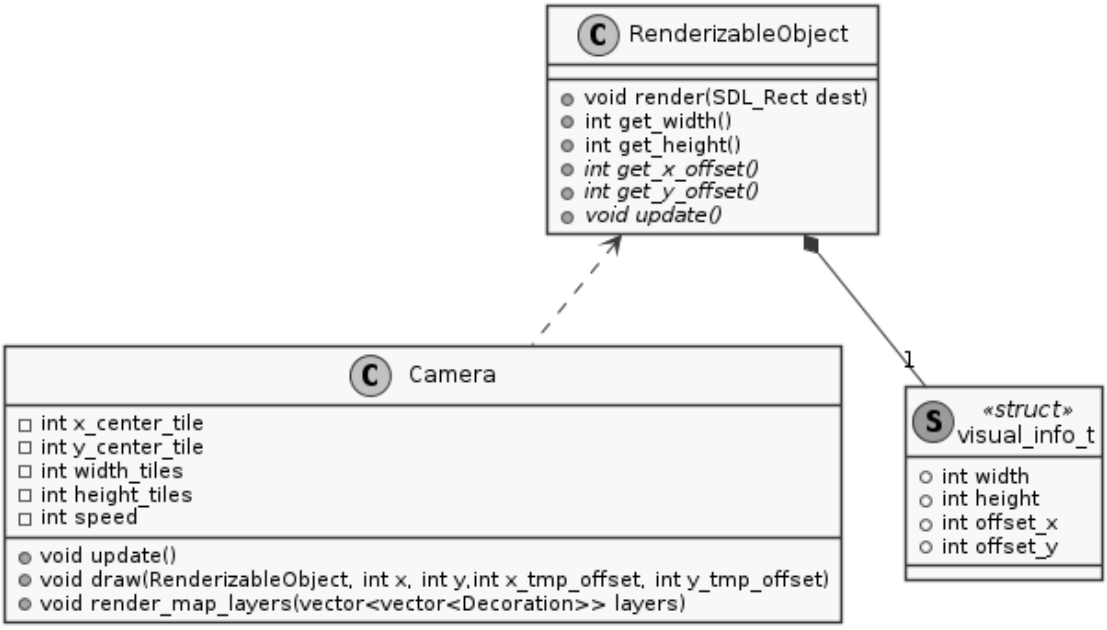

Figura 10: Diagrama de la camara.

6.2.2. Manager de assets.

Todos los assets (texturas, sprites animados, sonidos y fuentes) que el cliente necesita son contenidos y manejados por la clase ResourceManager, siendo estos cargados al comienezo de la ejecución del programa. Toda la información que el ResourceManager necesita para cargar estos assets a memoria está almacenada en un conjunto de índices en formato *json*, lo que hace que no sea necesario recompilar el cliente para cambiar una textura por otra, o para agregar nuevas texturas (o cualquier otro asset, como una fuente tipografica o un sonido).

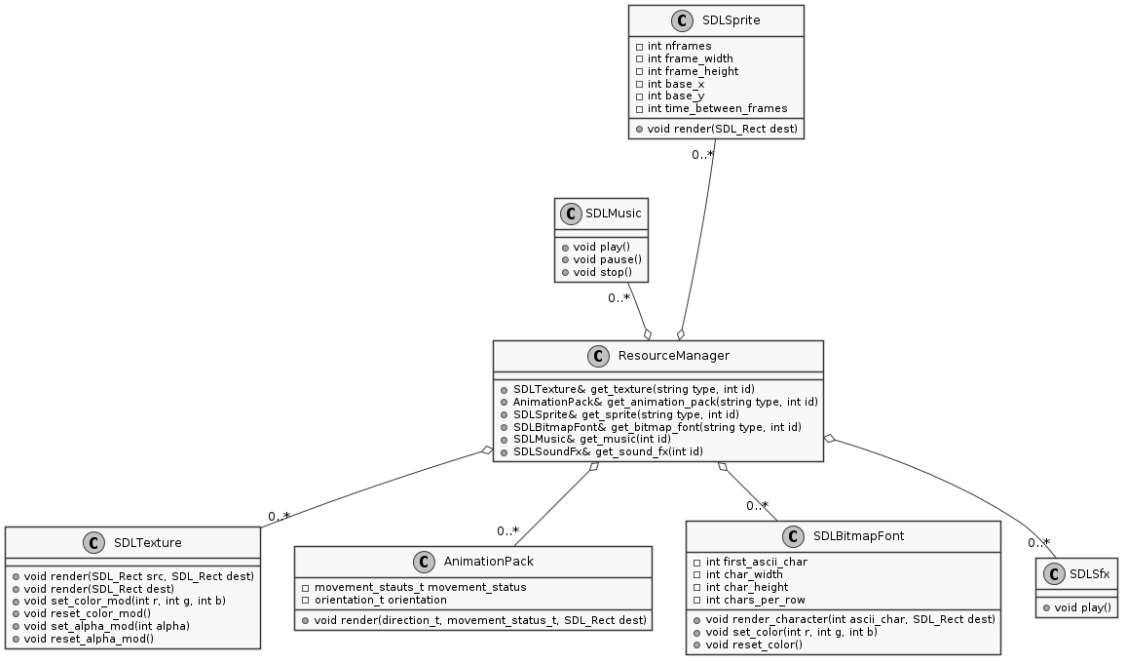

Figura 11: Diagrama de ResourceManager

El formato en el que se almacenan estos assets en memoria es mediante las primitivas de SDL que se mencionaron mas arriba.A continuación se detalla el formato en el que construyen los índices necesarios para cargar assets, los cuales se encuentran en el directorio *ind*.

6.2.2.1 Indexado de texturas

El indice de texturas corresponde al archivo con el nombre *textures.json*. Este archivo json tiene el siguiente formato:

```
{
    <categoria>:{
        "basedir": "assets/"<directorio base de la categoria>,
        "textures":[
            {
                "id":<id de la textura dentro de la categoria>,
                "filename": <nombre del archivo>,
                "needs ckey": <true/false>, (si se debe tomar un color como transparencia)
                "ckey" : [<r>,<g>,<b>] (color que se debe tomar como transparente)
            }
        ]
    }
}
```

Para agregar nuevas texturas, es suficiente con agregar nuevos items con este formato dentro de la lista de *textures* para la categoria a la que pertenece a la nueva textura. Alternativamente, tambien se puede agregar una nueva categoria respetando el formato indicado, dado que las categorias tambien se cargan dinamicamente a partir del indice.

6.2.2.2 Indexado de sprites animados

El indice de sprites corresponde al archivo con el nombre *sprites.json*. Dentro de este archivo, existen dos supercategorias: *Actors*, para aquellas animaciones que posean informacion sobre las cuatro direcciones, y *Decorations*, para aquellas que solo posean una direccion.

Los objetos dentro de *Actors* se indexan de la siguiente manera:

```
{
    "actors":{
        <categoria>:[
            {
                "id":<id>,
                "texture":<id de la textura base, cuya categoria debe coincidir con la del sprite>,
                "idle": { //Informacion sobre el estado 'quieto' del actor.
                    <"down"/"left"/"right"/"up">:{ //Se debe repetir esta
                    //informacion para las 4 direcciones.
                        "base x":<Coordenada x(pixeles) donde comienza la animacion en la textura>,
                        "base y":<Coordenada y(pixeles) donde comienza la animacion en la textura>,
                        "fps": <Velocidad de reproduccion de la animacion en fps>,
                        "frame height": <Alto(pixeles) de cada cuadro en la textura>,
                        "frame width": <Ancho(pixeles) de cada cuadro en la textura>,
                        "frames": <Cantidad de cuadros de la animacion>
                    }
                },
                "moving": { //Informacion sobre el estado 'moviendose' del actor.
                    <"down"/"left"/"right"/"up">:{ //Se debe repetir esta
                    //informacion para las 4 direcciones.
                        "base x":<Coordenada x(pixeles) donde comienza la animacion en la textura>,
                        "base y":<Coordenada y(pixeles) donde comienza la animacion en la textura>,
                        "fps": <Velocidad de reproduccion de la animacion en fps>,
                        "frame height": <Alto(pixeles) de cada cuadro en la textura>,
                        "frame width": <Ancho(pixeles) de cada cuadro en la textura>,
                        "frames": <Cantidad de cuadros de la animacion>
                    }
                }
            }
        ]
    }
}
```

Por otra parte, los objetos dentro de *Decorations* se indexan de la siguiente manera:

```
{
    "decorations":{
        <categoria>: [
            {
                "id":<id>
                "textura": <id de textura de la misma categoria>
                "base x": <Coordenada x(pixeles) donde comienza la animacion en la textura>,
                "base y": <Coordenada y(pixeles) donde comienza la animacion en la textura>,
                "fps": <Velocidad de reproduccion de la animacion en fps>,
                "frame height": <Alto(pixeles) de cada cuadro en la textura>,
                "frame width": <Ancho(pixeles) de cada cuadro en la textura>,
                "frames": <Cantidad de cuadros de la animacion>
            }
        ]
    }
}
```

6.2.2.3 Indexado de fuentes.

El indice de fuentes corresponde al archivo con el nombre *fonts.json*. Dentro de las fuentes, hay dos categorias: las *truetype* fonts, y las *bitmap* fonts.

Los objetos dentro de *truetype* se encuentran de la siguiente manera:

```
{
    "truetype":{
        "basedir":"assets/"<directorio base>,
        "fonts":[
            {
                "id":<id de fuente>,
                "filename":<nombre del archivo>
            }
        ]
    }
}
```

Y por otra parte, los objetos dentro de *bitmap* tienen esta forma:

```
{
    "bitmap":{
        "fonts":[
            {
                "id":<id de la fuente>,
                "texture_id":<id de la textura>
                "char_width":<ancho de cada caracter>,
                "char_height":<alto de cada caracter>,
                "first_ascii_char":<valor ascii del primer caracter de la textura>,
                "chars_per_row":<cantidad de caracteres por fila en la textura>
            }
        ]
    }
}
```

6.2.2.4 Indexado de sonidos

Los sonidos estan indexados en el archivo *audio.json*. Dentro de este archivo, hay dos categorias: *music* y *sfx*. Los objetos dentro de cada categoria tienen el mismo formato:

```
{
    "music":{
        "basedir":"assets/"<directorio base de la musica>,
        "files":[
            {
                "id":<id de la musica>,
                "filename":<nombre del archivo>
            }
        ]
    },
    "sfx":{
        "basedir":"assets/"<directorio base de los efectos de sonido>,
        "files":[
            {
                "id":<id del efecto de sonido>,
                "filename":<nombre del archivo>
            }
        ]
    }

}
```

6.2.3. Sistema de sonidos

Todos los sonidos del juego son manejados por la clase SoundSystem. Es una clase bastante simple, cuya unica responsabilidad es manejar la cantidad de sonidos reproduciendose en un tiempo dado. Para esto, consta con una cantidad n de sonidos que se pueden reproducir de forma simultanea. Si la cantidad de sonidos reproduciendose actualmente es la maxima, cualquier nuevo sonido que se intente reproducir sera simplemente ignorado. Cabe destacar que cada "slot"de sonido tiene un timeout arbitrario y constante, que *no es* la duracion del efecto de sonido reproduciendose en ese slot. Esto es para agregar simplicidad al codigo.

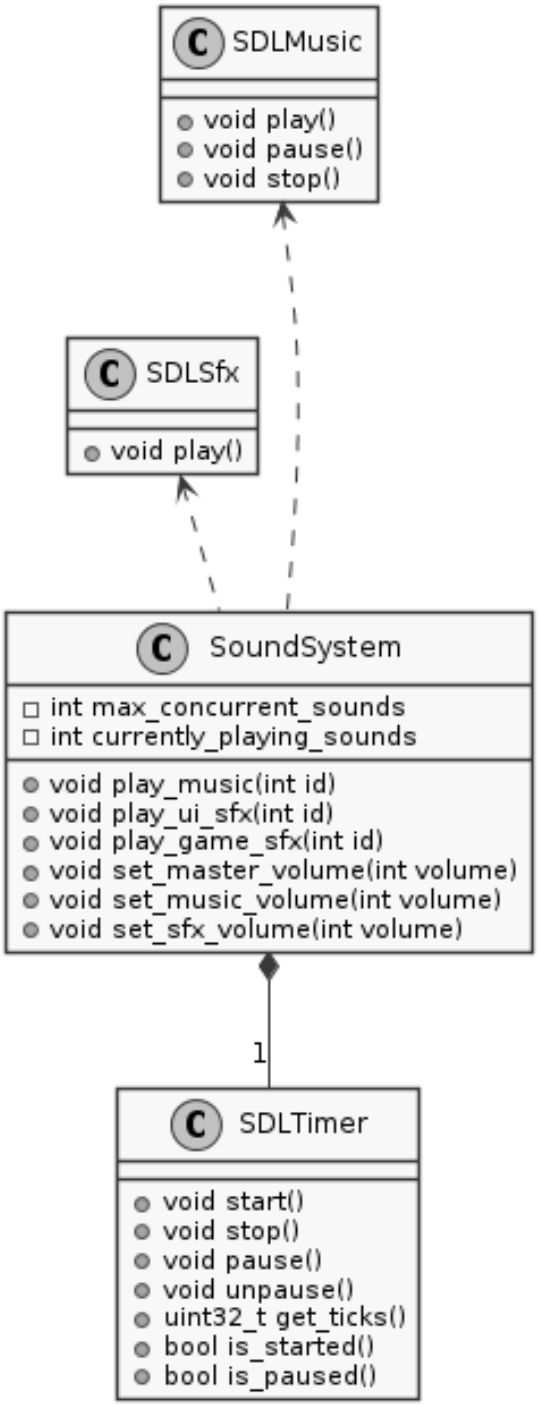

Figura 12: Diagrama de SoundSystem

6.2.4. Sistema de entidades y componentes

Todos los actores dinamicos del juego (jugadores y NPCs) son instancias de la clase Entity. Cada entidad, a su vez, está compuesta por un conjunto de Componentes.

Un jugador, por ejemplo, esta compuesto por un PositionComponent, encargado de mantener

la posicion del jugador; un VisualCharacterComponent, en el cual se encapsula toda la logica asociada al renderizado del jugador en la pantalla; un S oundComponent encargado de la reproduccion de sonidos asociados al jugador ; y por ultimo un StatsComponent, encargado de mantener informacion sobre vida, mana, nivel, etc. del jugador.

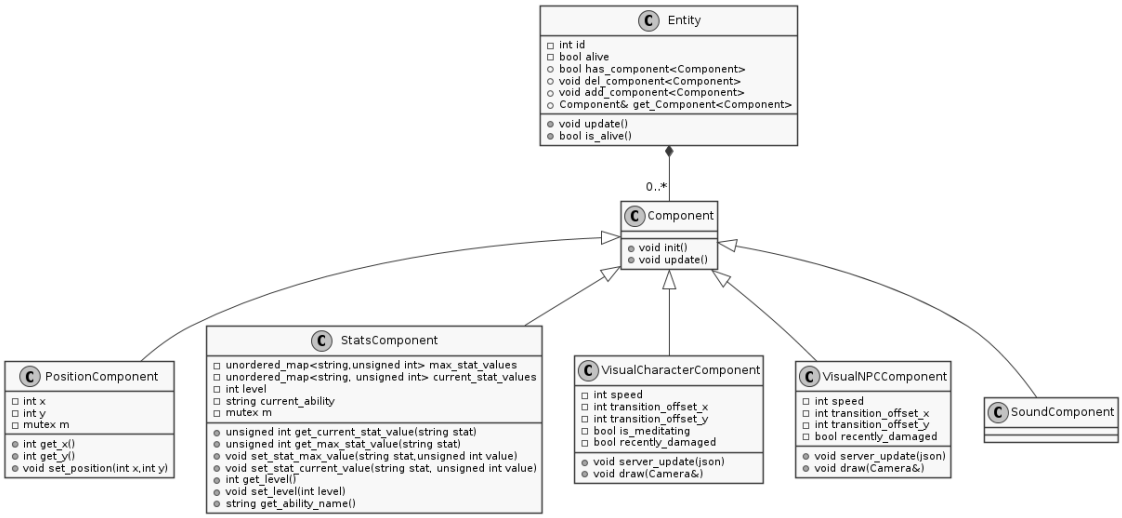

Figura 13: Diagrama del sistema de entidades y componentes

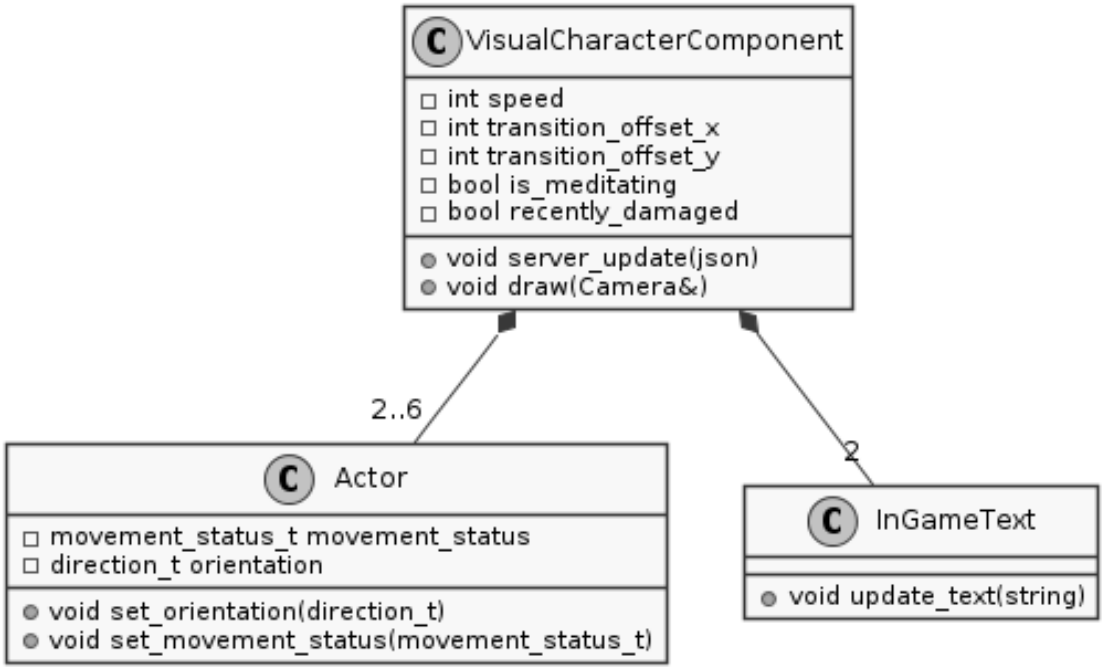

Figura 14: Detalle de VisualCharacterComponent

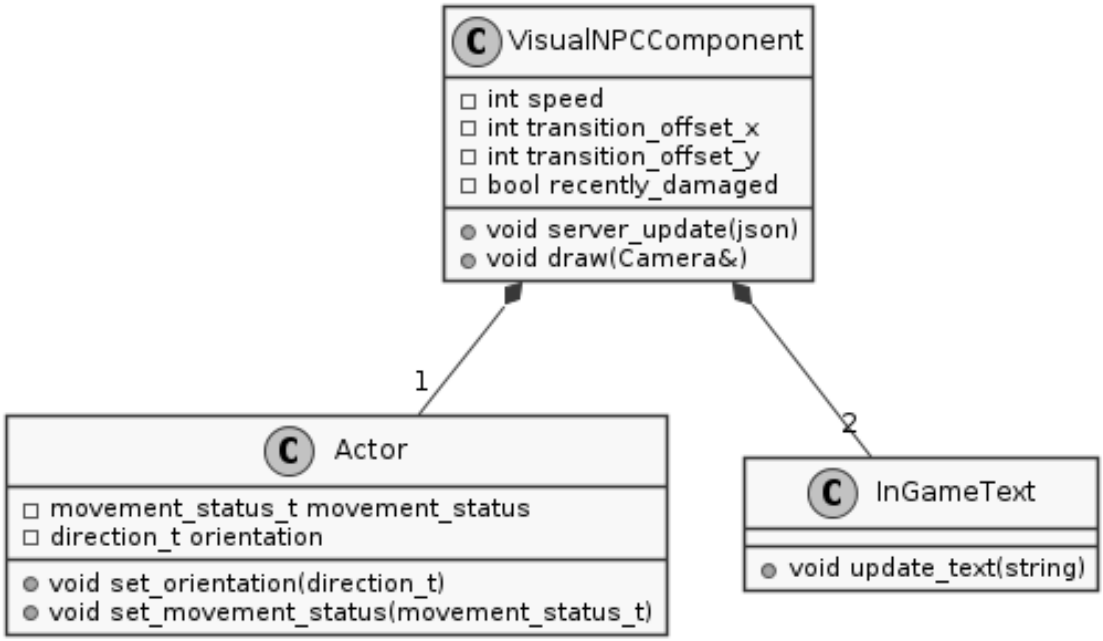

Figura 15: Detalle de VisualNPCComponent

6.2.5. Widgets de la UI

Para la construcción de la UI, se implementaron un conjunto de widgets básicos de la UI, que incluyen un Button, TextInput, una TextBox y una barra de "progreso"StatBar.
Todos los elementos de la UI tienen en comun la caracteristica de que son estaticos. Reciben al momento de su instanciación un área en la cual se renderizarán, la cual no puede cambiarse posteriormente.

Boton es una clase abstracta, que tiene dos metodos virtuales que deben ser escritos por las clases herederas: onClick y onRightClick, que son los llamados cuando se clickea el boton con click izquierdo, y cuando se lo clickea con click derecho (respectivamente). Para construir botones, solo es necesario crear una clase que herede de Boton e implemente estos metodos.

6.3. Sincronizacion y flujo de ejecucion

El flujo de ejecución del cliente (cambio entre vistas y sincronización con los mensajes del servidor) está orquestrado por el objeto `GameStateMonitor`. Este monitor encapsula un estado del juego (que es único para cualquier momento dado), en base al cual se determina cual es la siguiente vista que se debe mostrar.

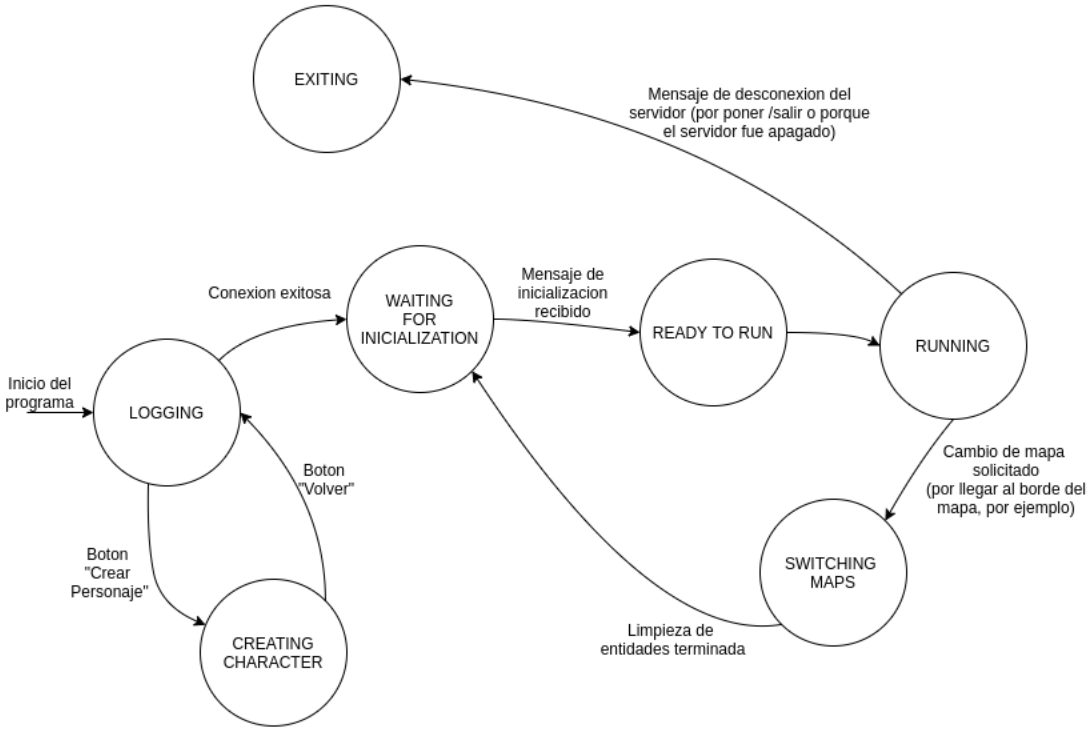

Figura 16: Cliente representado como maquina de estados finitos.

6.3.1. Estados del cliente

6.3.1.1 Estado LOGGING

Este estado del cliente es el primer estado que se establece al ejecutar el cliente. Indica que la vista que debe ser renderizada es la vista de inicio de sesión.

6.3.1.2 Estado WAITING_FOR_INICIALIZATION

Al iniciar sesión o cambiar de mapa, el cliente se pone en estado WAITING_FOR_INICIALIZATION, lo cual lo "bloquea"hasta que reciba el mensaje de inicialización del servidor. Como mecanismo de protección, si el servidor envia un mensaje de inicialización sin que el cliente esté en este estado, el thread de handleo de eventos provenientes del servidor se bloquea hasta que el cliente ingrese en este estado (Esto se debe a que, al cambiar de mapa, todas las entidades se borran de memoria, y se inicializa el nuevo estado del juego desde cero. Si el mensaje de inicialización llegara antes de que las entidades se terminen de borrar por completo, la entidad propia del jugador podría ser accidentalmente borrada instantáneamente luego de ser agregada).

6.3.1.3 Estado READY_TO_RUN

Indica que el mensaje de inicialización fue recibido correctamente, y que el juego se encuentra listo para empezar a correr en su estado jugable.

6.3.1.4 Estado RUNNING

Indica que el juego se encuentra corriendo en este momento, y es el estado que se tiene en todo momento en el que el jugador esté jugando propiamente el juego.

6.3.1.5 Estado SWITCHING_MAPS

Indica que el jugador solicito un cambio de mapa, y que por tanto se debe frenar la ejecución actual, borrar todas las entidades, y esperar un nuevo mensaje de inicialización por parte del servidor para el nuevo mapa.

6.3.1.6 Estado EXITING

Indica que, por alguna u otra razón, se está terminando la ejecución del cliente. Esto permite salir ordenadamente del juego.

6.3.1.7 Estado CREATING_CHARACTER

Estado que indica que se solicitó ir a la pantalla de creación de personaje.

6.4. Sistema de sincronizacion de informacion

Para que los cambios provenientes del servidor se materialicen en lo que ve el cliente de forma ordenada, se utiliza un patron *Double Buffering*, implementado mediante un conjunto de buffers que actúan de monitores para la información que proviene del server, que luego son accedidos por el thread del Game Loop al momento de actualizar las entidades del juego.

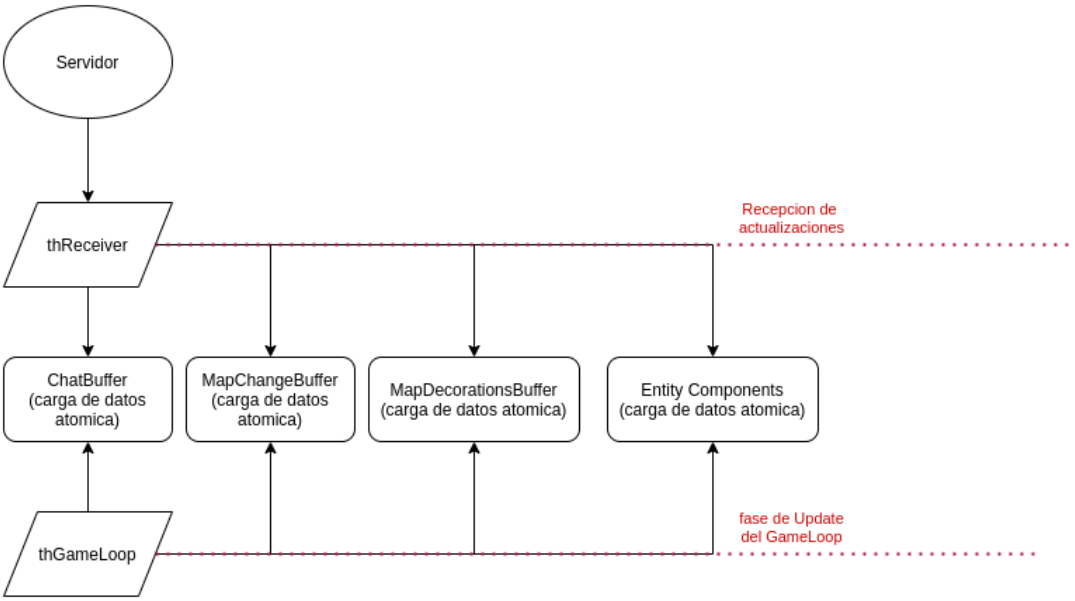

Figura 17: Diagrama general de la aplicación.

7. Servidor

El servidor fue construido para funcionar de forma concurrente para varios jugadores que estuviesen conectados en múltiples mapas al mismo tiempo, para ello se dispuso a construir la parte lógica del juego para que corriera de forma independiente y se construyó el sistema de comunicación con los clientes por encima de ésta. Es inicializado por el thread principal a partir de la clase `GameServer` que espera el comando para cerrar el servidor por la entrada estándar. Esta clase inicializa el objeto que orquesta el resto de los threads y componentes del servidor: `ServerManager`.

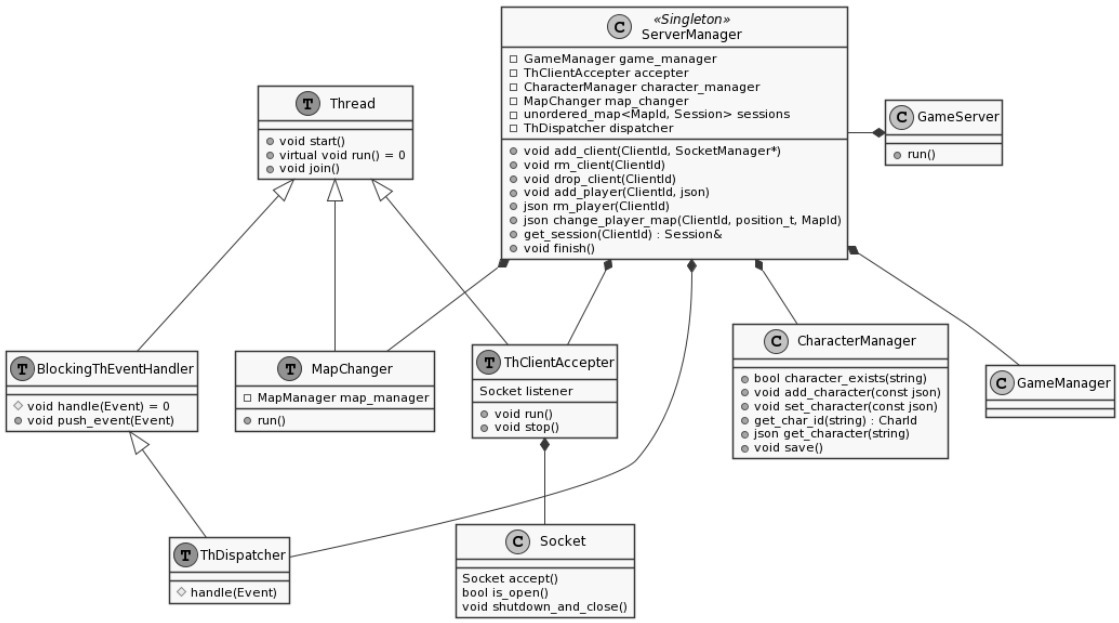

Figura 18: Diagrama de clases principal del servidor.

7.1. Eventos entrantes

Una de las componentes principales del servidor es el `Dispatcher`, un Thread encargado de obtener todos los eventos que deber realizar el servidor y despacharlo al objeto que haga el manejo correspondiente. Los objetos que permiten realizar el *"handle"* de un evento son hijos de la clase `EventHandler`, a partir de la cual también hereda la versión *Threaded* de la misma (`BlockingThEventHandler`).

Los eventos que requieren interacción con la lógica del juego lo logran a través de acciones en el mapa, son despachadas a él a través de cada sesión, que traduce Ids de clientes a Ids de entidades que conoce el mapa.

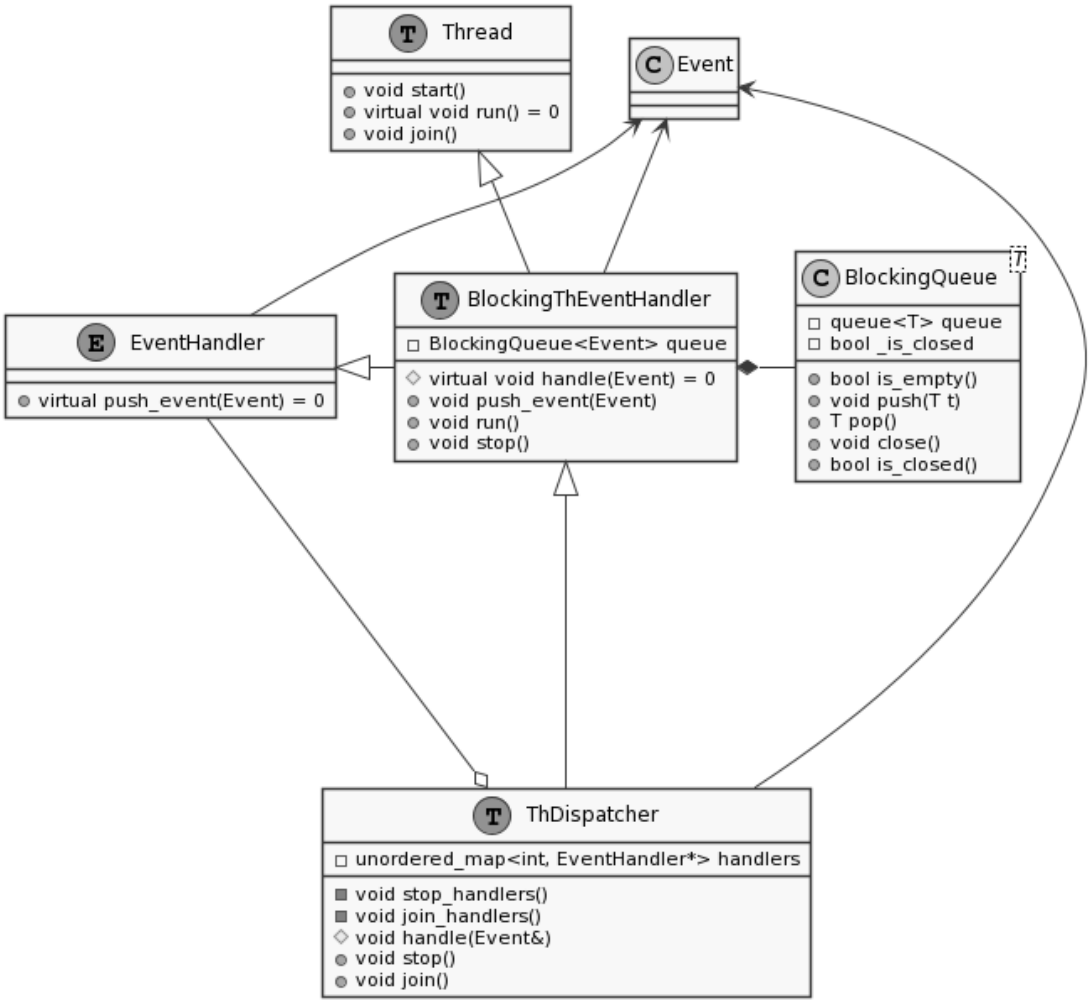

Figura 19: Diagrama de clases de EventHandler.

7.2. Lógica del Juego

La lógica del juego se encuentra aislada de la comunicación y Eventos. Es controlada por una clase principal: `GameManager`. Este se encarga de iniciar el `GameLoop`, el thread encargado de mantener actualizada la lógica del juego, indicando al contenedor de mapas `MapManager` que actualice cada uno de los mapas que contiene.

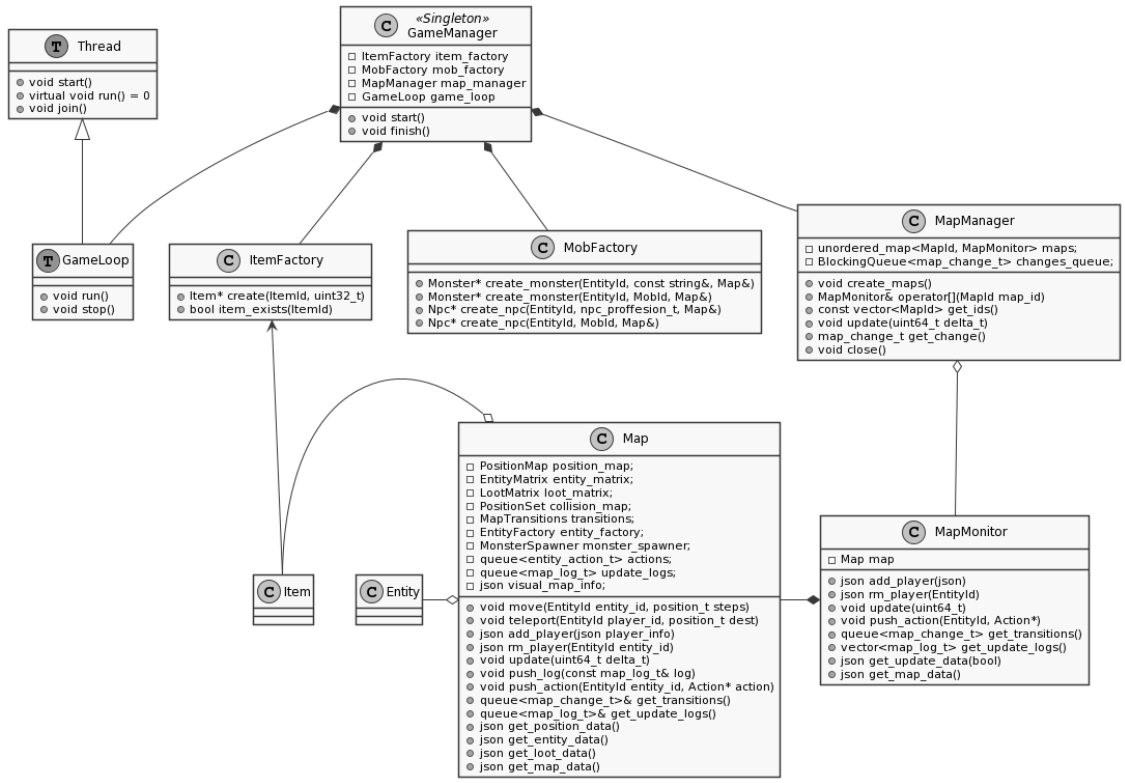

Figura 20: Diagrama de clases del GameManager.

Los mapas son las estructuras que mantienen las posiciones lógicas de las `entidades` (`Player`, `Npc` y `Monster`) y `objetos` tirados en el mapa. Además en cada actualización ejecuta las `acciones` agregadas a cada mapa y actualiza las entidades según un *time-step* que recibe del `GameLoop`. Para poder extraer información de los mapas se provee la posibilidad de extraer la información de las entidades y sus posiciones, así como registros de acciones particulares de cada jugador (ya sean acciones como ataques o interacciones con el inventario de su jugador).

7.2.1. Entidades y sus componentes

A continuacion se adjuntan diagramas de clase de las entidades mencionadas que componen la logica del juego

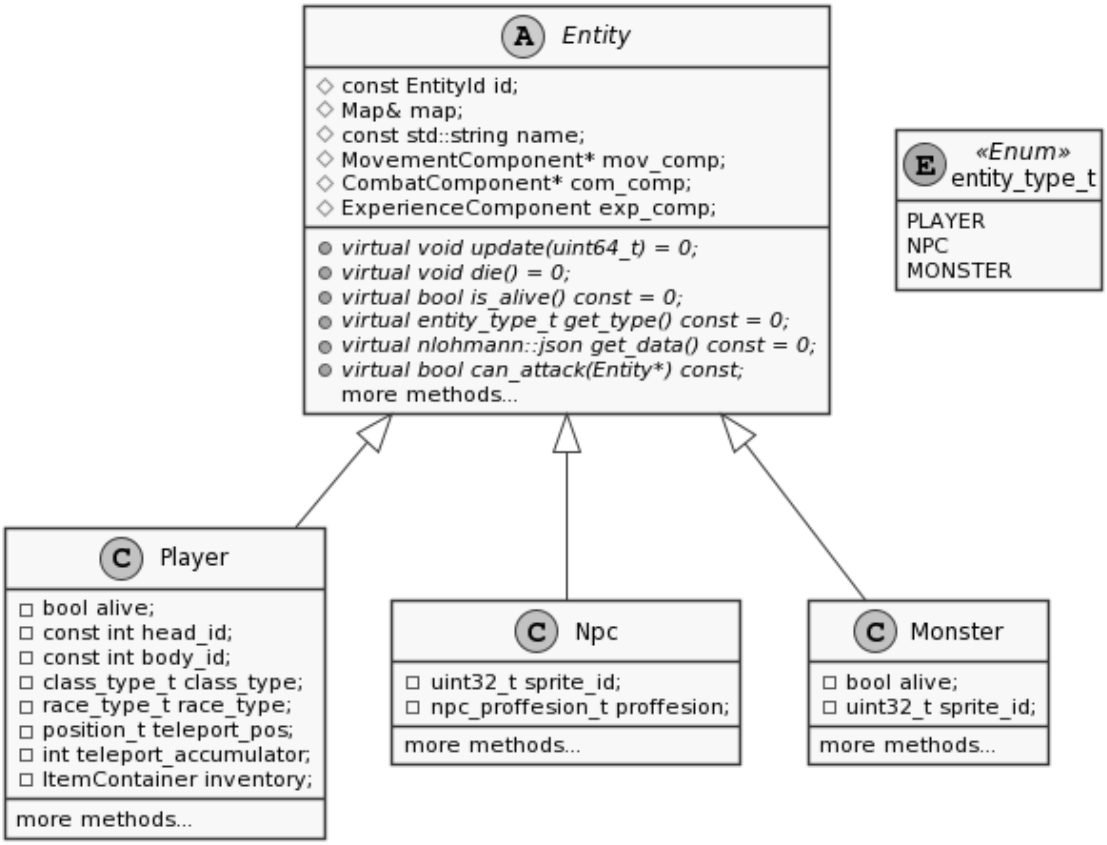

Figura 21: Diagrama general de Entity

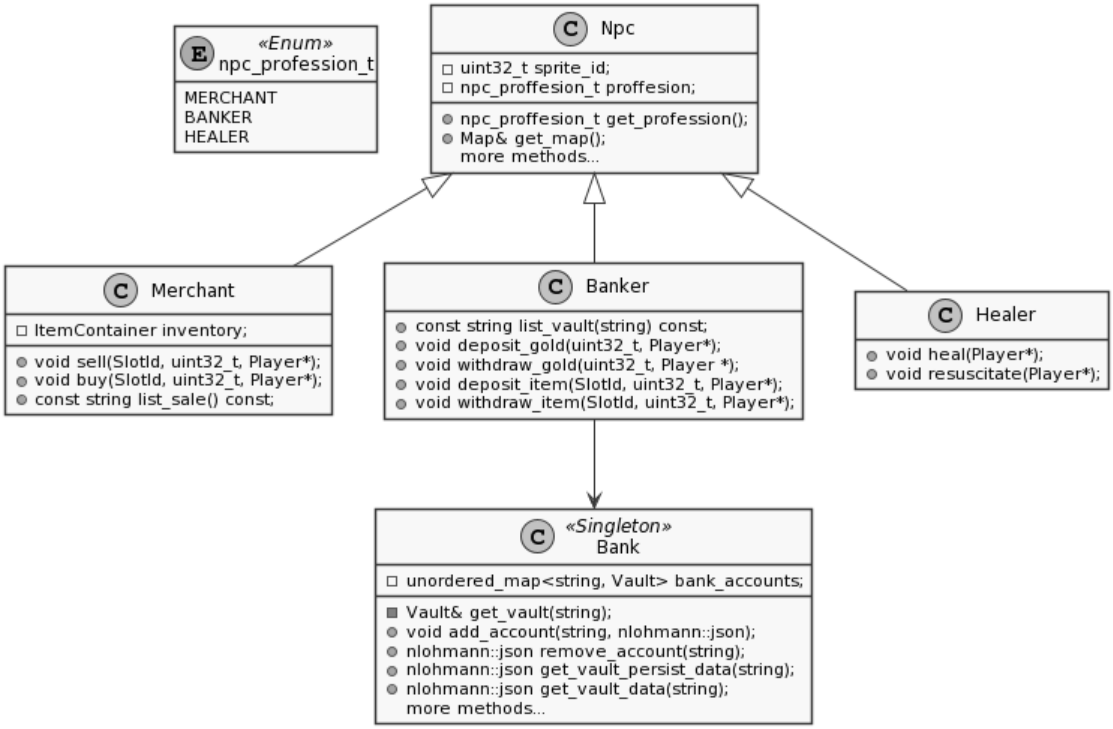

Figura 22: Diagrama general de Npc's

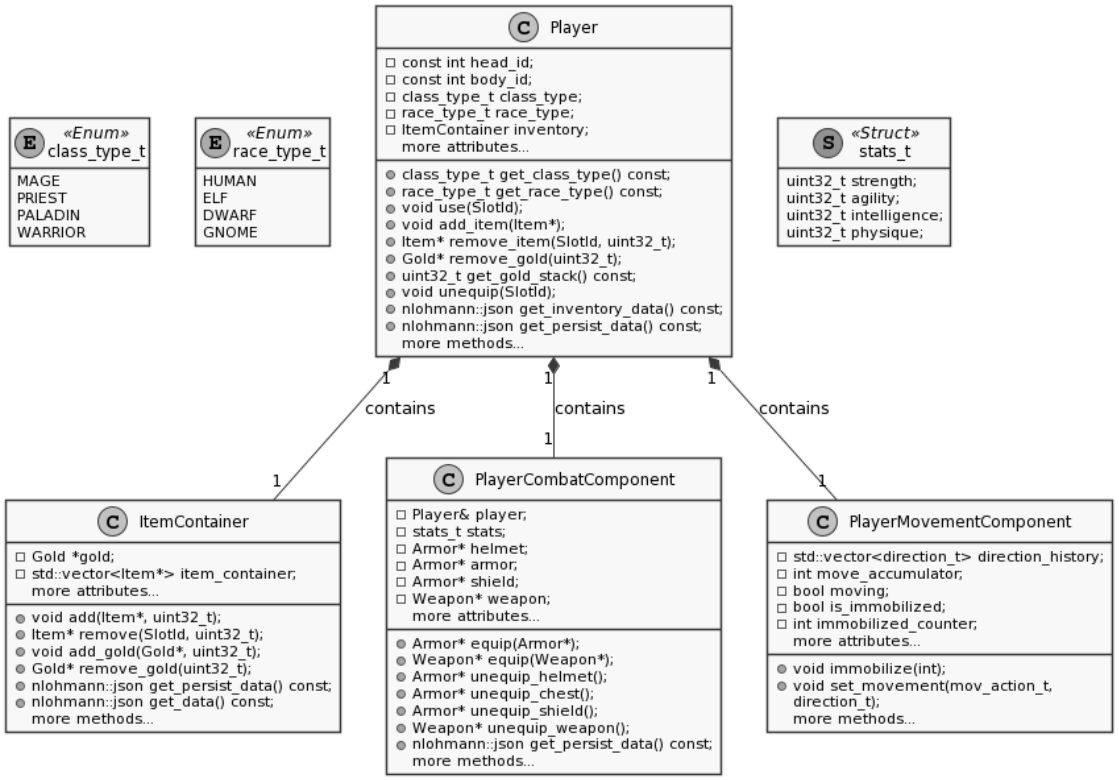

Figura 23: Diagrama general de Player

Como se puede verificar en el diagrama de clases del jugador, las entidades estan compuestas por un conjunto de componentes que encapsulan los comportamientos logicos de los distintos sistemas (el inventario, el movimiento, y el combate). Esta misma idea se extiende a todas las entidades: El comportamiento de las entidades esta definido por los componentes que las componen.

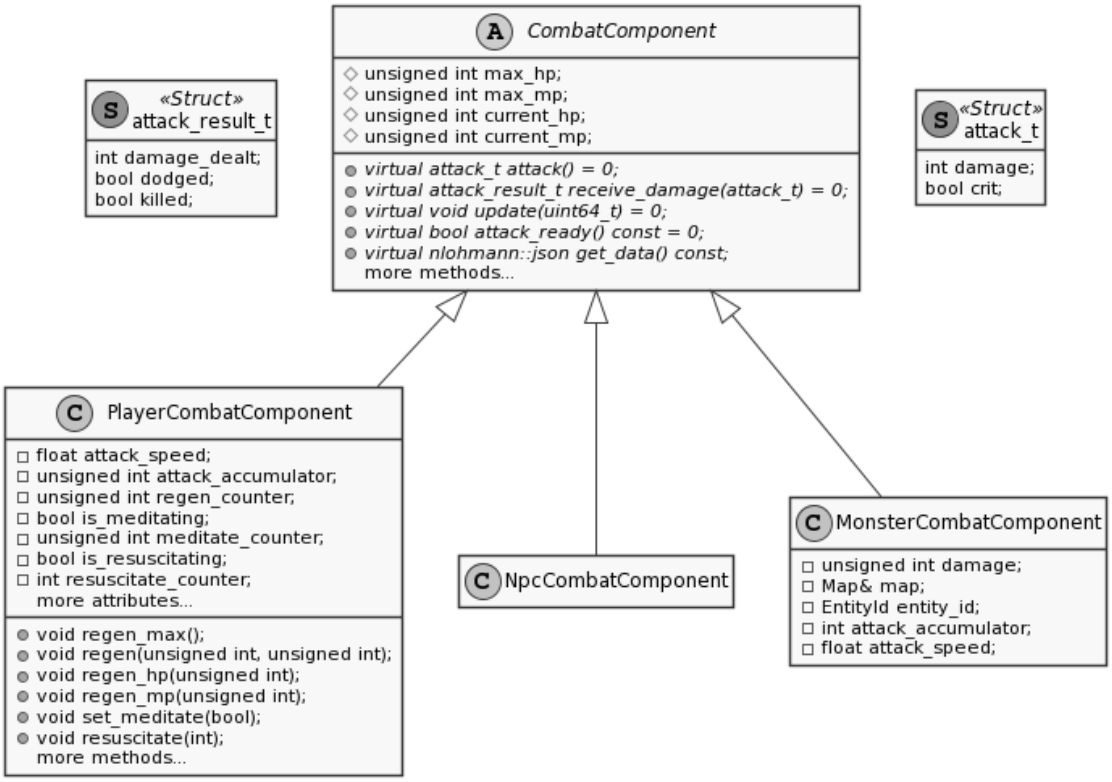

Figura 24: Diagrama general de CombatComponent

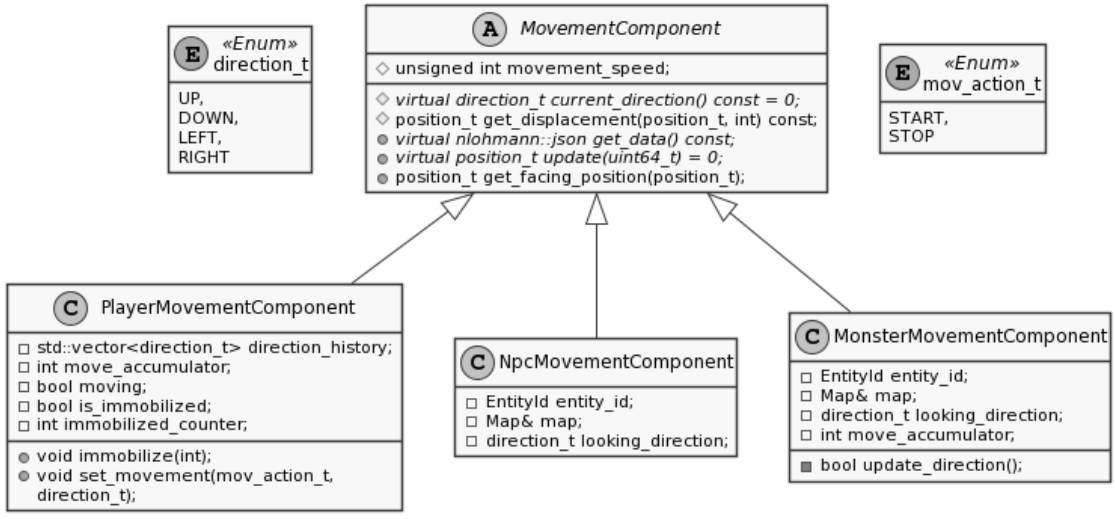

Figura 25: Diagrama general de MovementComponent

7.2.2. Objetos

Todos los objetos del juego heredan de una clase Item. Existen tres principales clases que heredan de la misma: Weapon, Armor y Potion.

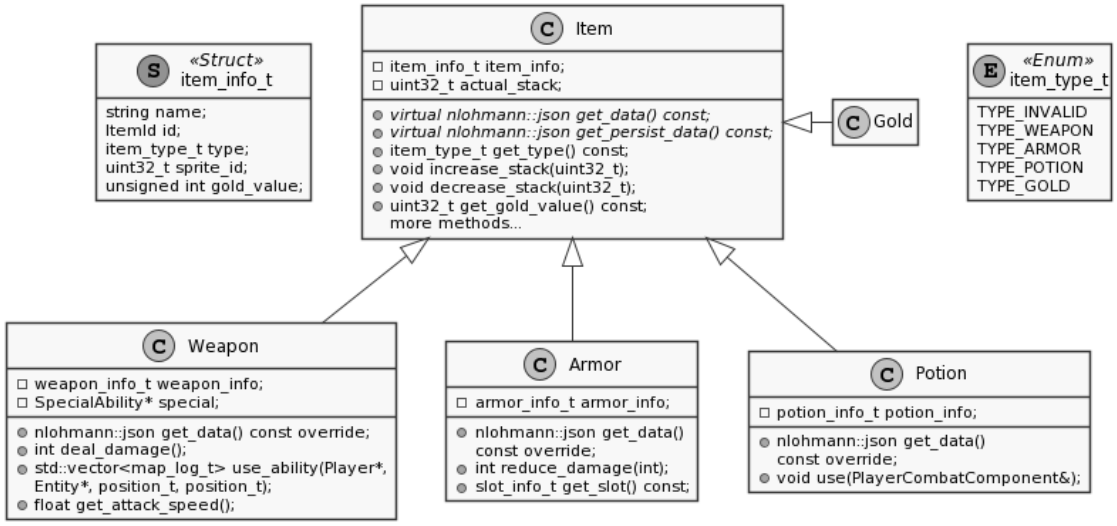

Figura 26: Diagrama general items

Dado que todas las armas tienen el mismo comportamiento (salvo ataques especiales), se decidio que todas sean parametrizaciones de la misma clase Weapon, con especificaciones acorde al arma.

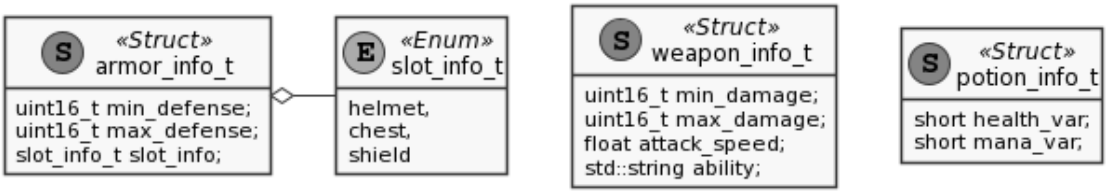

Figura 27: Informacion especifica de los distintos tipos de items

7.3. Polling de cada mapa

Por cada mapa el servidor cuenta con un objeto Session que mantiene la comunicación del servidor con cada mapa dado. Por cada una de estas sesiones se tienen dos *Threads*: `Broadcaster` (permite enviar a todos los jugadores conectados a la sesión un mismo evento) y `Observer` (encargado de realizar el polling del mapa). Este último realiza lecturas periódicas del mapa leyendo además los registros dejados para los jugadores particulares.

7.4. Persistencia

Para la persistencia se utilizó un objeto `CharacterManager` que maneja estructuras (`c struct`) especializadas para ser guardadas en un archivo de tipo objeto, con un archivo `json` para indexarlo.

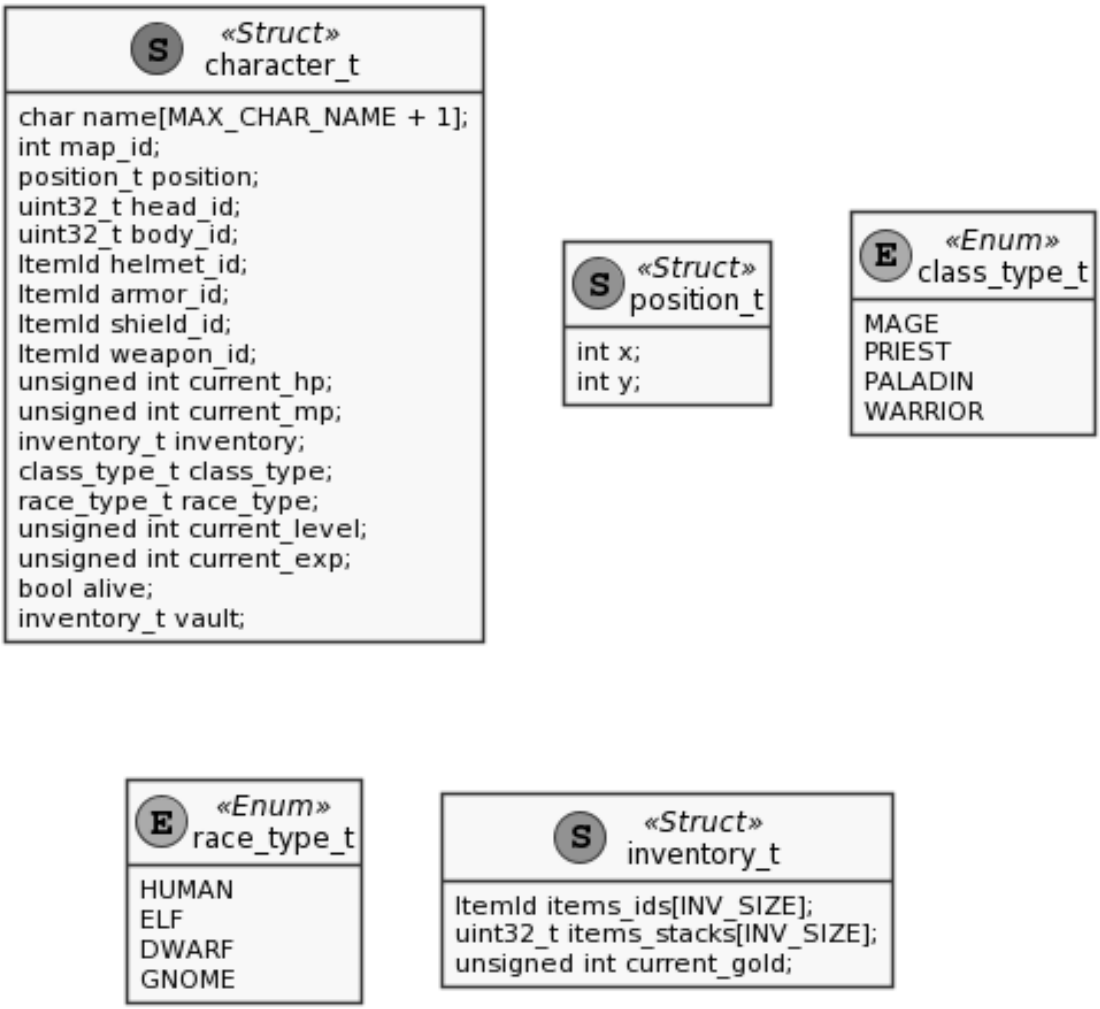

Figura 28: Estructuras de la persistencia

8. Comunicación

De ambos lados del programa (cliente y servidor), se emplea una clase `SocketManager` para el manejo de la comunicación concurrente a través de los sockets.

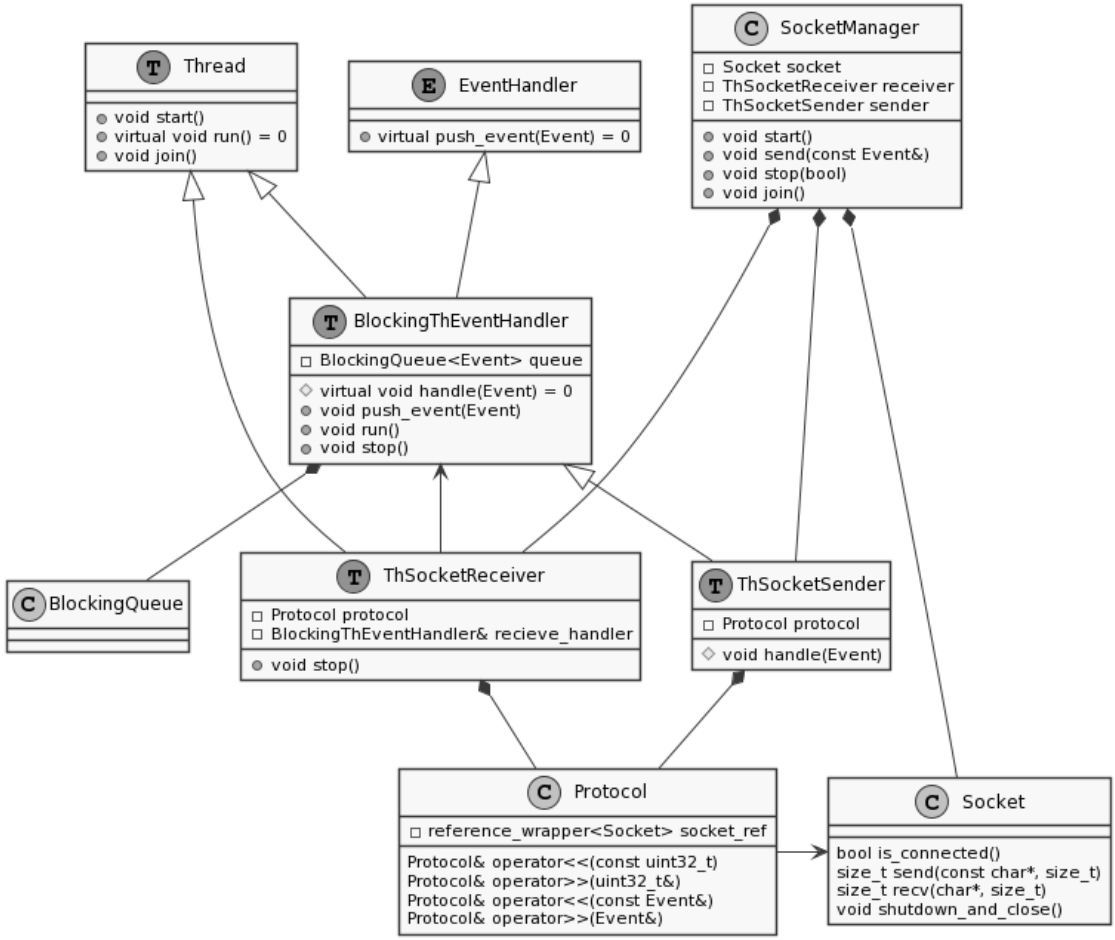

Figura 29: Diagrama de SocketManager.

Una vez ejecutado el servidor, éste estará esperando conexión de nuevos clientes, que pueden iniciar sesión de dos formas diferentes: creando un jugador o entrando con un jugador existente (ver los mensajes correspondientes a continuación). Cuando un cliente ya está conectado y su jugador fuera agregado al mapa, recibirá el mensaje para inicializar el mapa seguido de un mensaje que actualiza todas las entidades y objetos tirados del mismo. Una vez hecho esto, el servidor le envía periódicamente mensajes que actualizan las posiciones de las entidades y cada algunas de estas también se envía información de las entidades y objetos caídos en el mapa. A continuación también se describen los mensajes para desconexión de un cliente.

8.1. Cliente → Servidor

Para al comenzar el cliente, iniciar sesión con un personaje nuevo se envía la siguiente información.

```
{
    "ev_id": EV_ID_CREATE,
    "client_id": <ClientId>,
    "name": <string>,
    "class_type": <class_type_t>,
    "race_type": <race_type_t>
}
```

Para iniciar sesión con un personaje ya existente, el cliente envía el siguiente mensaje al servidor.

```
{
    "ev_id": EV_ID_CONNECT,
    "client_id": <ClientId>,
    "player": {
        "name": <string>,
        "password": <string> // No utilizado
    }
}
```

Al presionar o levantar una de las teclas de movimiento, el cliente envía la información correspondiente en el siguiente mensaje.

```
{
    "ev_id": EV_ID_MOVE,
    "client_id": <ClientId>,
    "movement": {
        "action": <mov_action_t>,
        "direction": <direction_t>
    }
}
```

Al presionar la tecla para ataque (control), el cliente envía la siguiente información al servidor.

```
{
    "ev_id": EV_ID_ATTACK
    "client_id": <ClientId>,
}
```

Al presionar la tecla 'a' (agarrar) el cliente envía lo siguiente al servidor.

```
{
    "ev_id": EV_ID_PICKUP_LOOT
    "client_id": <ClientId>,
}
```

Al hacer click derecho sobre una posición del inventario, el cliente envía la siguiente información al servidor.

```
{
    "ev_id": EV_ID_DROP_LOOT,
    "client_id": <ClientId>,
    "slot": <SlotId>
}
```

Al hacer doble click (izquierdo) sobre una posición del inventario, el cliente envía la siguiente información al servidor.

```
{
    "ev_id": EV_ID_INVENTORY,
    "client_id": <ClientId>,
    "slot": <SlotId>
}
```

Al escribir cualquier mensaje en la consola, se envía la información dispuesta a continuación. Notar que se envía información de dónde se hizo click (izquierdo) sobre el mapa por última vez y qué *"Slot"* del inventario se hizo click (izquierdo) por última vez.

```
{
    "ev_id": EV_ID_COMMAND,
    "client_id": <ClientId>,
    "msg": <string>,
    "slot": <SlotId>,
    "target": {
        "x": <int>,
        "y": <int>
    }
}
```

Cuando un cliente se quiere desconectar y notificar al servidor de ello, envía el siguiente mensaje.

```
{
    "ev_id": EV_ID_DISCONNECT
    "client_id": <ClientId>,
}
```

8.2. Servidor → Cliente

Mensaje de inicialización. Este se envía cada vez que se agrega un jugador a un mapa, esto quiere decir que sucede cada vez que un cliente inicia sesión (o crea un personaje) y cuando un jugador cambia de mapa.

```
{
    "ev_id": EV_ID_INITIALIZE_MAP,
    "client_id": <ClientId>, // Siempre será 0
    "map_info": {
        //
        // Información gráfica del mapa, extraida
        // de un archivo de salida del progrma Tiled
        //
    },
    "player": {
        // player_init_data
        //
        // player_entity_data (Ver entity_data)
        //
        "pos": {
            "x": <int>,
            "y": <int>
        },
        "inventory": {
            //
            // inventory_data
            //
        }
    }
}
```

Mensaje de actualización gráfica de las entidades. Es enviado periódicamente a los clientes (con menos frecuencia que el que actualiza las posiciones de las entidades, contiene mucha más información).

```
{
    "ev_id": EV_ID_UPDATE_ENTITIES,
    "client_id": <ClientId>, // Siempre será 0
    "entities": [
        {
            //
            // entity_data:
            "entity_id": <EntityId>,
            "name": <string>,
            "direction": <direction_t>,
            "move_speed": <uint>,
            "type_id": <entity_type_t> ,
            "curr_hp": <uint>,
            "max_hp": <uint>,
            "curr_mp": <uint>,
            "max_mp": <uint>,
            "curr_level": <uint>,
            "curr_exp": <uint>,
            "limit_exp": <uint>,
            //
            // entity_player_data:
            "type_id": PLAYER,
            "head_id": <uint>,
            "body_id": <uint>,
            "helmet_id": <uint>,
            "armor_id": <uint>,
            "shield_id": <uint>,
            "weapon_id": <uint>
            //
            // entity_monster_data:
            "type_id": MONSTER,
            "sprite_id": <uint>
            //
            // entity_npc_data:
            "type_id": NPC,
            "sprite_id": <uint>

        },
        ...
    ]
}
```

Mensaje de actualización del *"loot"*, es decir, los objetos que son dejados en el piso del mapa.

```
{
    "ev_id": EV_ID_UPDATE_LOOT,
    "client_id": <ClientId>, // Siempre será 0
    "items": [
        {
            //
            // item_data
```

```
            //
        },
        ...
    ]
}
```

Mensaje de actualización de posiciones de entidades sobre el mapa, es enviado a cada paso de actualización del *Observer*.

```
{
    "ev_id": EV_ID_UPDATE_MAP,
    "client_id": <ClientId>, // Siempre será 0
    "positions": [
        {
            "entity_id": <EntityId>,
            "x": <int>,
            "y": <int>
        },
        ...
    ]
}
```

Mensajes para cuando un jugador hace daño a un objetivo o recibe daño de otra entidad.

```
{
    "ev_id": EV_ID_DEALT_DAMAGE,
    "client_id": <ClientId>, // Siempre será 0
    "dmg": <int>,
    "to": <EntityId>
}

{
    "ev_id": EV_ID_RECEIVED_DAMAGE,
    "client_id": <ClientId>, // Siempre será 0
    "dmg": <uint>
}
```

Mensaje del chat o consola para el cliente.

```
{
    "ev_id": EV_ID_CHAT_MESSAGE,
    "client_id": <ClientId>, // Siempre será 0
    "msg": <string>
}
```

Mensaje de actualización de inventario para un cliente. Este mensaje es privado, es decir, se envía únicamente al cliente controlando al jugador propietario del inventario.

```
{
    "ev_id": EV_ID_INVENTORY_UPDATE,
    "client_id": <ClientId>, // Siempre será 0
    "inventory": {
        "curr_gold": <uint>,
        "items": [
            {
                // 1era posición del inventario.
```

```
                "type": <item_type_t>
                //
                // Si type != TYPE_INVALID
                // item_data:
                "actual_stack": <uint>,
                "item_id": <ItemId>,
                "name": <string>,
                //
                // Ej: TYPE_POTION
                //
                "potion_info": {
                    "health_var": <short>,
                    "mana_var": <short>
                },
            },
            {
                // 2da posición del inventario.
                "type": <item_type_t>
            },
                ...
            {
                // 12ava posicion del inventario.
                "type": <item_type_t>
            }
        ]
    }
}
```

Mensaje para notificar a un cliente que se enviará un mapa nuevo, esto sucede cuando un jugador es cambiado de mapa.

```
{
    "ev_id": EV_ID_NOTIFY_NEW_MAP,
    "client_id": <ClientId> // Siempre será 0
}
```

Mensaje informativo a un cliente para notificar que ha sido desconectado del servidor.

```
{
    "ev_id": EV_ID_DROP_CLIENT,
    "client_id": <ClientId> // Siempre será 0
}
```