Universidad de Buenos Aires

Facultad de Ingeniería

75.42 - Taller de Programación

Lenguaje: C++

Primer cuatrimestre de 2023

Manual Técnico

Bravo, Nicolas Francisco	106753	nbravo@fi.uba.ar
Chávez Cabanillas, José E.	96467	jchavez@fi.uba.ar
Davico, Mauricio	106171	mdavico@fi.uba.ar

Índice

1. Introducción

El presente trabajo realizado se basa en Left 4 Dead un videojuego de disparos en primera persona, cooperativo y asimétrico del género «videojuego de terror», desarrollado por Turtle Rock Studios y publicado por Valve Corporation. Para este trabajo se simplificaron muchas de las mecánicas, para que se pudiera realizar en 2D.

2. Requerimientos del Sistema

El proyecto usa varias librerías para poder ejecutarse, así como también de un sistema operativos basado en alguna distribución GNU/Linux, a continuacíon se mencionaran las librerías necesarias para poder compilar y ejecutar el juego.

2.1. SDL 2

Esta librería es el motor gráfico del juego, se encarga de todo lo relacionado con la renderización, musicalización e interacción con el usuario (Cliente), determinando cada acción del usuario para luego mostrarla en pantalla.

A su vez esta librería cuenta con librerías adicionales, que complementan y forman parte de toda el motor gráfico. Estas librerías son las de SDL2 Mixer y SDL2 TTF, encargándose de la reproducción de la música del juego, y la visualización de fuentes respectivamente.

Figura 1: SDL

2.2. Qt5

Qt5, es una librería para trabajar con interfaz gráfica, la ventaja de esta librería es que utiliza el lenguaje de programación `C++` de forma nativa, lo que cual fue una gran ventaja para el proyecto, cuenta con diversos módulos.

Dentro de estos módulos, para el proyecto se requirieron el moódulo de Widgets y de Multimedia, siendo el primero para la creación de la interfaz, mediante el uso de buttons, labels, lineEdits, etc. El segundo, se encarga de la música de fondo entre los diversos menús del cliente.

Figura 2: Qt

2.3. YAML-cpp

El nombre de la librería nos da una pista sobre el uso de esta, su funcionalidad radica en el parser de archivos YAML, el cual se usa para cargar la configuración del juego.

3. Descripción general

El trabajo cuenta con dos programas ejecutables, un cliente y un servidor, estos tendrán interacción por medio de un protocolo de comunicación basado en sockets TCP/IP.

En el contexto de una aplicación o juego, el cliente cumple un papel fundamental al gestionar la comunicación con el usuario en la parte visible y accesible de la interfaz (front-end). Esto implica que el cliente no solo recibe y procesa los eventos que ocurren, sino también las acciones que el usuario realiza. Por otro lado, el servidor asume la responsabilidad de manejar toda la lógica del juego o la aplicación (back-end), respondiendo de manera adecuada a las acciones que el usuario lleva a cabo.

A continuación se puede observar una vista de alto nivel del esquema de comunicación adoptado:

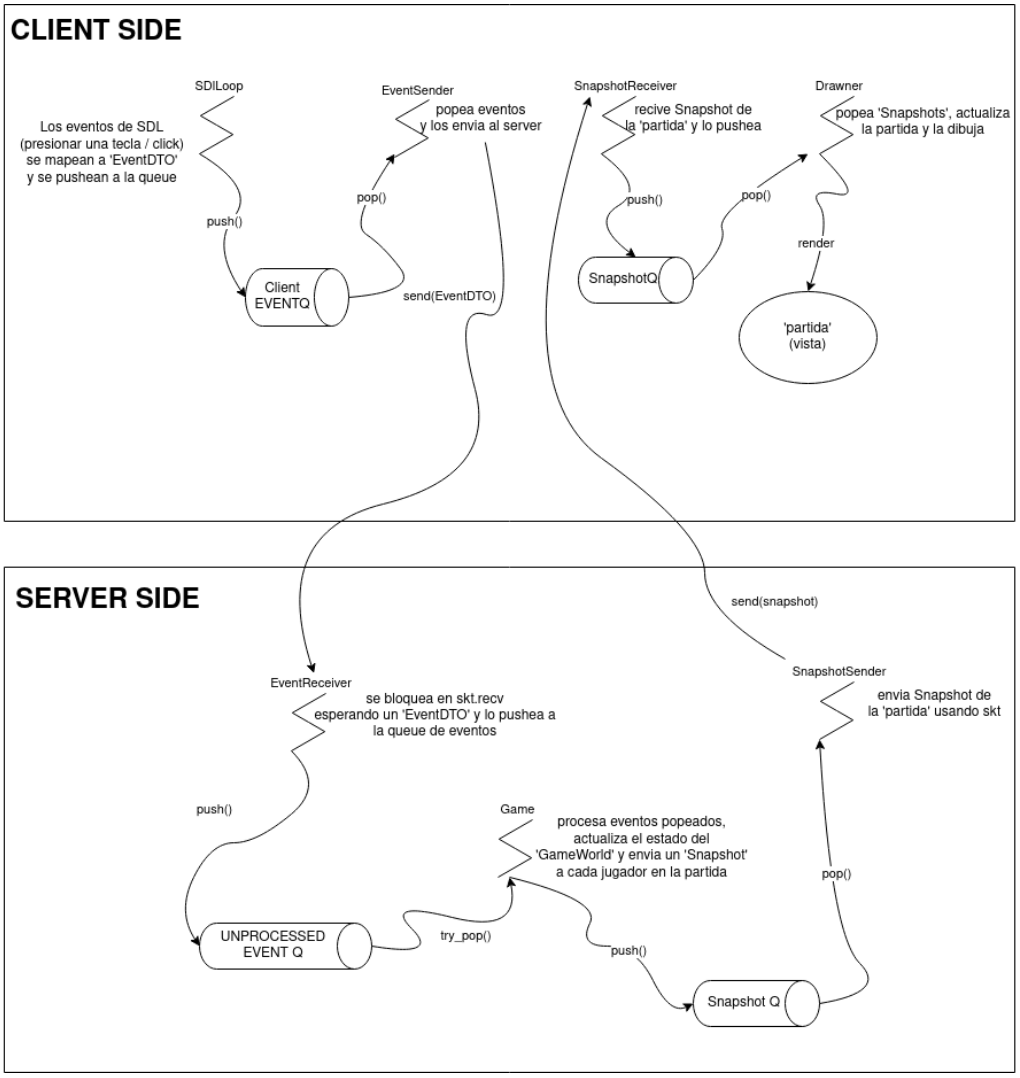

Figura 3: Hilos de Comunicación

Gracefull Quit:

- Cliente: Se busca siempre cerrar la conexión del socket en el hilo EventSender, para esto, ante cualquier error o flujo esperado de cierre, se deberá cerrar la queue de eventos.
- Servidor: Se busca siempre cerrar la conexión desde el hilo Snapshor Sender, para esto, ante

cualquier error o flujo de cierre esperado, se deberá cerrar la queue de snapshots (de los clientes que correspondan).

EventReceiver - Game: En el servidor, al iniciar el hilo EventReceiver se va a encontrar en un estado que podemos llamar "lobby", el cual se caracteriza simplemente por que el hilo Game aun no se a iniciado. En esta estado se procesan los create / join / leave en el mismo hilo EventReceiver. Una vez que el hilo Game a iniciado, y miestras este no haya terminado, todos los eventos recibidos se pushea a la queue de eventos, para que sean procesados en el hilo Game.

4. Módulo Común

4.1. Descripción General

En este apartado se encuentran detallados aquellos recursos que son utilizados, para distintos fines, tanto en el cliente como en el servidor. Luego, se detalla como se llevaron a cabo los tests.

4.2. Recursos Compartidos

4.2.1. Brindados por la cátedra

Las siguientes clases fueron implementación de la cátedra, brindadas a los alumnos para su utilización en el presente trabajo.

- Queue: Utilizado para diversos fines.
  - Queue de eventos detectados en el cliente.
  - Queue de snaphots recibidos en el cliente.
  - Queue de eventos recibidos en el servidor (todos a ser procesador por hilo Game).
  - Queue de snapshots a enviar hacia el cliente.
- ClosedQueue: Excepción de Queue.
- Socket: Utilizado para la comunicación.
- LibError: Excepción de Sokcet.
- Resolver: Utilizado por Socket.
- ResolverError: Excepción de Resolver.
- Thread: Utilizado para diversos fines.
  - GameDrawner - cliente.
  - EventSender - cliente.
  - SnapshotReceiver - cliente.
  - EventReceiver - servidor.
  - SnapshotSender - servidor.
  - Acceptor - servidor.
  - Game - servidor.

4.2.2. Comunicación cliente - servidor

Para la implementación del protocolo, se definió una clase **Protocol** donde se encuentran los métodos comunes. El protocolo implementado recibe siempre como argumento un puntero al socket a utilizar. Luego, en cada modulo cliente - servidor se encuentran sus respectivas implementaciones, con los métodos exclusivos de cada modulo, que heredan de la clase común antes mencionada.

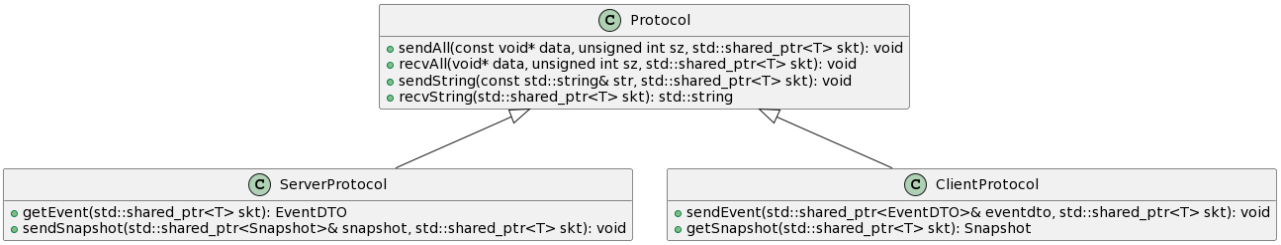

Figura 4: Diagrama de clases Protocol

Para la comunicación entre cliente - servidor se definieron algunas clases que cumplen la función de DTOs, detallados a continuación:

- EventDTO: Es el objeto que representa los eventos generados por el jugador del lado del cliente, ya sea utilizando el teclado o presionando botones en la interfaz gráfica. Se utiliza como argumento del método ClientProtocol::sendEvent() el cual se encarga de enviarlo por medio del socket al servidor. Del lado del servidor, el método ServerProtocol::getEvent() es quien se encarga leer el socket y construir el EventDTO correspondiente, para que el servidor pueda diferenciar y procesar los eventos generados en el cliente.

- Snapshot: Lo que representa este objeto va a depender del contexto donde se utilice:

  - **El juego aun no comenzo, los clientes se encuentran en el lobby**: Contiene las respuestas a los eventos de create/join.

  - **El juego ya comenzó**: Contiene la información necesaria para que el modulo cliente renderize el estado actual de juego.

  Similar a lo que sucede con los EventDTO, los Snapshot se pasan como argumento al método ServerProtol::sendSnapshot() para que se envie por medio del socket al cliente, y desde el modulo del cliente se reciben llamando al método ClientProtocol::getSnapshot().

A su vez, tanto EventDTO como Snapshot utilizan otros DTO y enums definidos para poder diferenciar que evento / estado de juego se debe representar, estos son:

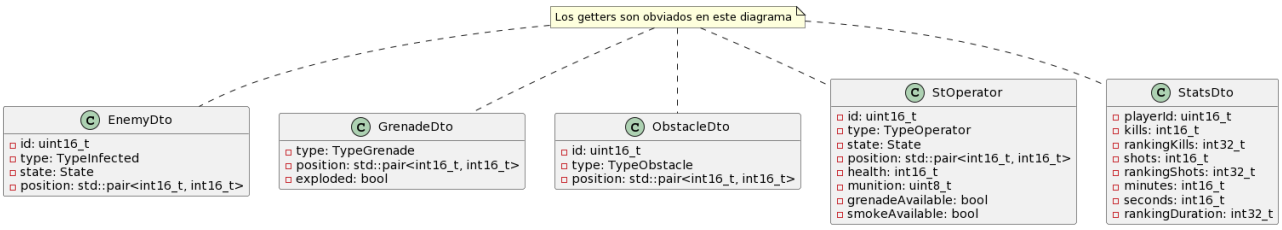

Figura 5: Diagrama de clases DTOs

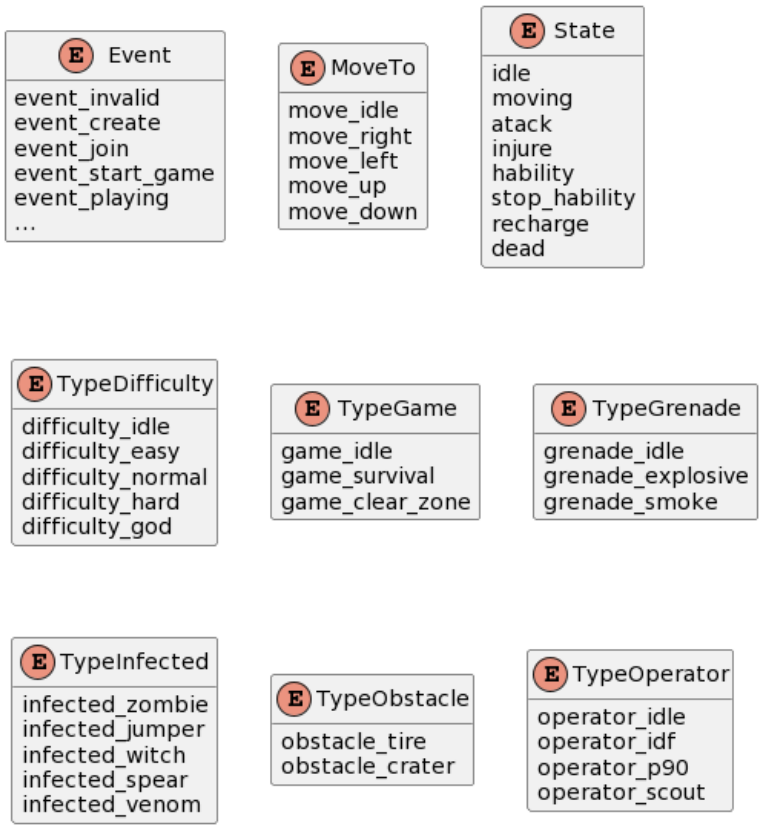

Figura 6: Diagrama de clases Enums

4.3. Tests

Para realizar los test primero se estudio cual era el Framework mas completo, luego de analizar las opciones existentes, se tomo la decisión de utilizar GoogleTest.

A continuación se detallan los módulos testeados:

- Protocolo: Aprovechando que el protocolo recibe el socket como un puntero a un template, se implemento la clase **SimulatedSocket** para utilizar como mock, el mismo simula la implementación brindada por la cátedra, con la diferencia que escribe y lee bytes de un array. En la implementación de los tests, se instancia un **SimulatedSocket**, que sera pasado tanto al **ClientProtocol** como **ServerProtocol**, para que escriban / leen bytes según corresponda. Se encuentran testeados todos los casos de usos posibles, distinguidos por tipo de evento, o por estado del juego, ya sea lobby o jugando, tanto en la comunicación desde el modulo cliente hacia el modulo servidor, y viceversa

- Colisiones: Se testean métodos públicos de la clase **Collidable**. Para ello se instancian los objetos necesarios y se evalúan las respuestas de los métodos de la clase.

- Estadísticas: Se definió un path auxiliar para realizar los tests, con el fin de no invalidar el archivo que mantiene las estadísticas reales del juego. En la implementación de los test se tiene en cuenta que la clase **StatsController** es quien lee el archivo en su constructor, mantiene ordenadas las estadísticas mientras exista, y persiste las estadísticas al archivo en su destructor.

Si bien no se siguió una metodología TDD, los tests, una vez implementados, fueron un soporte

fundamental a la hora de realizar cambios sobre los módulos antes detallados, ya que al automatizar tests sobre dichos módulos, no es necesario ejecutar el proyecto y llegar a cada caso de uso para verificar si los cambios funcionan correctamente.

5. Módulo Cliente

5.1. Descripción General

El módulo del cliente se encarga de establecer la conexión con el Servidor, gestionar los eventos al Servidor y procesar los snapshots recibidos por el Servidor. Todo a través de su interfaz gráfica, la cual consta de cuatro partes: Launcher, Lobby, GameSdl y EndGame.

La primera de estas partes fue elaborada con Qt5, las partes restantes fueron elaboradas con SDL.

5.2. Launcher

Como se mencionó anteriormente el Launcher se construyó mediante el uso de la librería de Qt5, esta parte es llamada así por que todos los eventos principales hasta el inicio de las siguientes partes es gestionada por la clase Launcher, la cual podemos denominar un gestor de ventanas, estas ventanas tienen un fin específico desde la conexión con el Servidor, la creación de una partida asi como también el poder unirse a una partida.

La interacción entre el gestor de ventanas (Launcher) y el resto de ventanas se hacen a través de SIGNALS (señales), las cuales son emitidas desde dichas ventanas y recibidas por el Launcher, el cual procesa la señal emitida y ejecuta la acción correspondiente.

```
void ConnectView::connectToServer() {
    QString ip = lineIp.text();
    QString port = linePort.text();
    if (!ip.isEmpty() && !port.isEmpty()) {
        Q_EMIT createConnection(ip, port);
    } else {
        QMessageBox::information(this,
            "Error",
            "Complete los datos correctamente",
            QMessageBox::Close);
    }
}
```

Código 1: Envio de una señal

El código anterior ”Envio de una señal”, ejemplifica el envió de una señal, para este caso al presionar el botón de conectar, la cual chequea que los QLineEdit donde se carga la información no estén vacíos, en caso de estarlos muestra un mensaje de error informando que los campos están vacíos, y si no lo están envía dicha información a través de la señal, la cual es recibida por la clase Launcher, para ejecutar algún método específico de la siguiente manera:

```
connect(&connectView, &ConnectView::createConnection, this,
&Launcher::createProtocol);

void Launcher::createProtocol(  const QString& ip,
                                const QString& port) {
    try {
        this->socket = std::make_shared<Socket>(
                                    ip.toStdString().c_str(),
                                    port.toStdString().c_str());
        mainWidget.setCurrentIndex(2);
```

```
    } catch (std::exception &exc) {
        QMessageBox::information(this,
                "Error",
                "Datos ingresados no validos",
                QMessageBox::Close);
    }
}
```

Código 2: Recepción de una señal

Como se aprecia en el código Recepción de una señal, la recepción consta de 2 dos partes, la primera de ellas es la detección de la señal emitida, siendo que clase la envía, que señal se emitió, a quien llego la señal, y que se debe hacer con dicha señal. Para esta última parte es el método que se aprecia, la cual crea y establece la conexión con el Servidor.

Con esto en mente la clase Launcher tiene todo los métodos centralizados en ella lo cuál genera una gran responsabilidad para todo la aplicación del cliente en su primera parte, cabe destacar que todo el Launcher esta acompañado de una música de fondo, todo gracias a uno de los submódulos de la librería llamada Qt5Multimedia, la cual permite agregar desde un directorio todo tipo de archivos musicales para poder reproducir.

5.3. Lobby

Una vez que el cliente crea o se une a una partida no iniciada, se pasa a la segunda parte del cliente, que es Lobby, a pesar de ser algo sencillo en lo visual, tiene una diferencia importante al saber si la persona que espera, es quien creo la partida o se unió a la partida.

Esta diferencia, permite al cliente que crea la partida tener el poder de iniciar la partida al generar un evento cuando se presione la tecla "Enter". Este evento es enviado al servidor para que sea procesado, para lo cual el servidor informará a todos los clientes conectados a la partida del comienzo de la misma.

Por otra parte el cliente que se une a la partida no puede generar ningún evento, debe esperar hasta que el creador de la partida de inicio. Para esta parte del cliente también se tiene música de fondo, sin embargo para toda esta parte desde su inicio hasta su final, el inicio de la partida, ya se empieza a usar la libreria de SDL, para lograr todo esto se usan SDL, SDL2_ttf, SDL2_mixer, el primero sirve para la creación de la pantalla render e inicialización de componentes, el segundo sirve para la utilización de fuentes y poder renderizarlas en toda la pantalla, y el último para la música de fondo durante toda la espera.

5.4. GameSdl

Terminada la etapa de Lobby, para cualquier caso de los mencionados anteriormente (se crea o se una a una partida), empezará la etapa del juego. En esta etapa se recibe la configuración inicial de la partida, ya sea para el modo **Clear Zone** o el modo **Survival**.

El gameloop principal de GameSdl, es básicamente:

```
    while (gameSdl.isRunning()) {
        uint32_t frameInit = SDL_GetTicks();

        render.clear();
```

```
        SDL_PumpEvents();
            gameSdl.update();
            gameSdl.handleEvents();
            gameSdl.render();
        render.present();

        uint32_t frameEnd = SDL_GetTicks();
        uint32_t processTime = frameEnd - frameInit;

        if (1000 / 40 > processTime)
            SDL_Delay(1000 / 40 - processTime);
    }
```

Código 3: Gameloop

En cada loop del juego, el render se encarga de limpiar la pantalla y mostrar todos los render que estarán presentes en la pantalla, el Método SDL_PumpEvents(), se asegura de mantener actualizados todos los eventos que ocurran en el juego, para nuestro caso el teclado. Las partes importantes son el update, handleEvents, render.

5.4.1. Update

Antes de cada renderizado en la pantalla, el servidor envía Snapshots del estado actual de la partida, los elementos a actualizar en cada loop son los elementos de la Operator, Enemy y Grenade.

Para los enemigos hay que tener en cuenta el modo de juego, si estamos en el caso de **Clear Zone**, bastara con chequear el id del enemigo se encuentre en la lista de GameSdl, en caso de no encontrarse se activará el estado de Muerte **(State::dead)**. Para el caso de encontrarse en el modo de juego **Survival** ese deberá tener en cuenta la cantidad de Updates realizadas, si la dicha cantidad resulta ser cero, sumado al tipo de juego, se puede afirmar que se esta en la condiciones de crear nuevos enemigos para el juego.

Para las granadas, se actualiza la posición del movimiento del lanzamiento, hasta su activación la cual setea en **true** la activación de la animación de explosión.

Para los operadores, se actualiza la posición, el estado en el que se encuentran, la cantidad de ráfagas disponibles, la vida, y la disponibilidad de habilidades que tiene disponible.

5.4.2. Handle Event

El Handle Events, es una de las partes más importantes del gameloop, se encarga de la detección de los eventos que ocurren en cada gameloop, detecta las teclas presionadas por el cliente y determina el evento que ocurre, para el cual arma un mensaje en un **EventDTO** el cual es pusheado en la Queue de eventos.

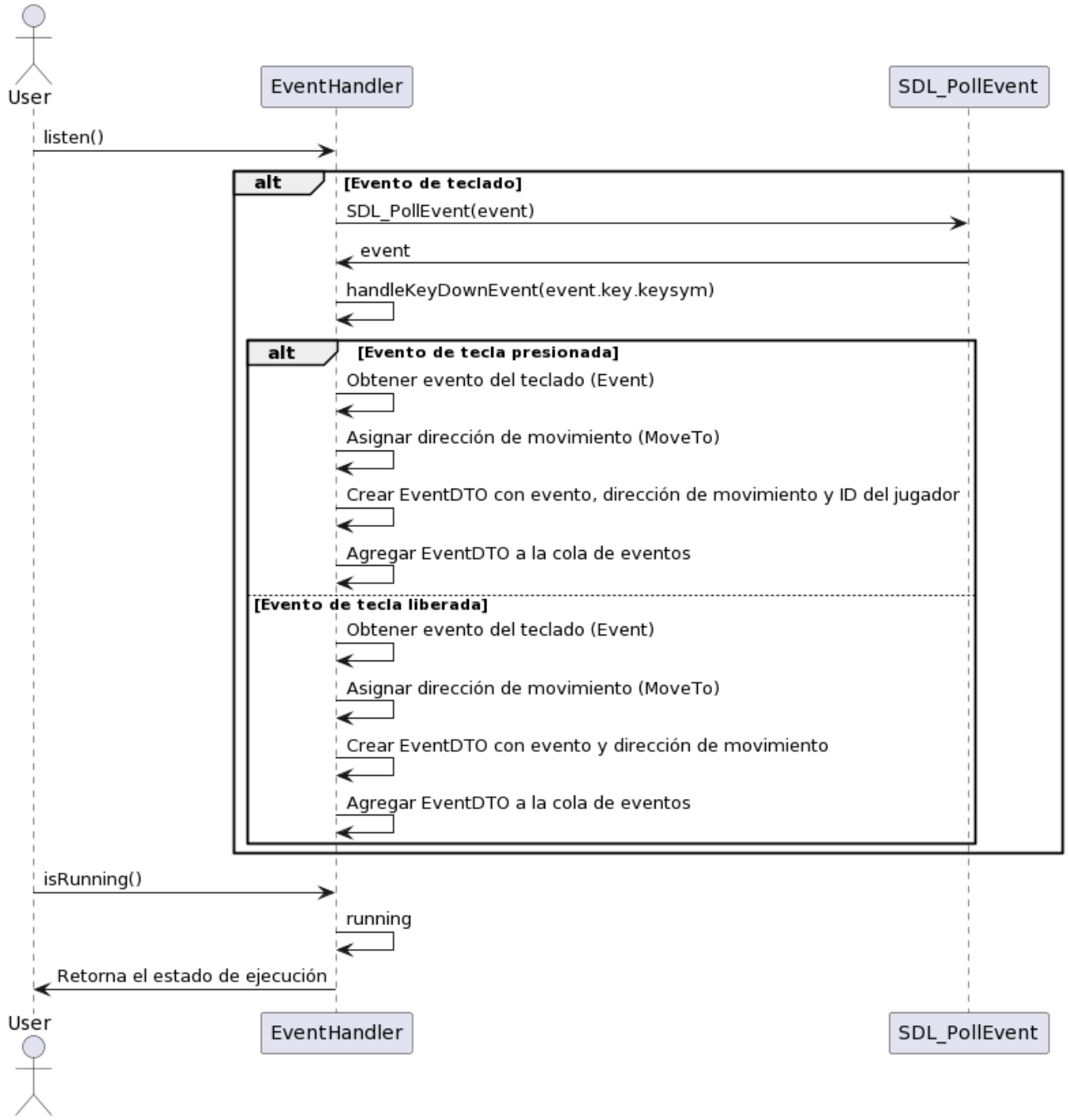

Figura 7: Diagrama de secuencia del Handle Event

5.4.3. Render

El render se encarga de colocar todas las piezas a ser dibujadas en la pantalla, cabe destacar que siempre se tiene que hacer en un orden específico para que cada pieza pueda ser visualizada correctamente.

El orden especifico en el que se renderizan los objetos son: Mapa, Hud, Operadores, Enemigos, Obstáculos, Granadas o habilidades, y en caso de estar en la etapa EndGame, renderiza el final especifico para el modo de juego.

También hay que mencionar que los Operadores, Enemigos y obstaculos siempre están presentes en el Mapa, lo cual hace que el renderizado se un poco más específico, dado que se necesita previo al renderizado reordenarlos por la posición en el eje Y en el que se encuentran, empezando desde el que tiene menor posición, con lo cuál se logra un efecto de profundidad con el entorno.

Este efecto se logró haciendo que las clases Enemy, Operator y Obstacles, sean clases heredadas de la clase Object, el cual permitió que se puedan reordenar y renderizar como un conjunto de todo.

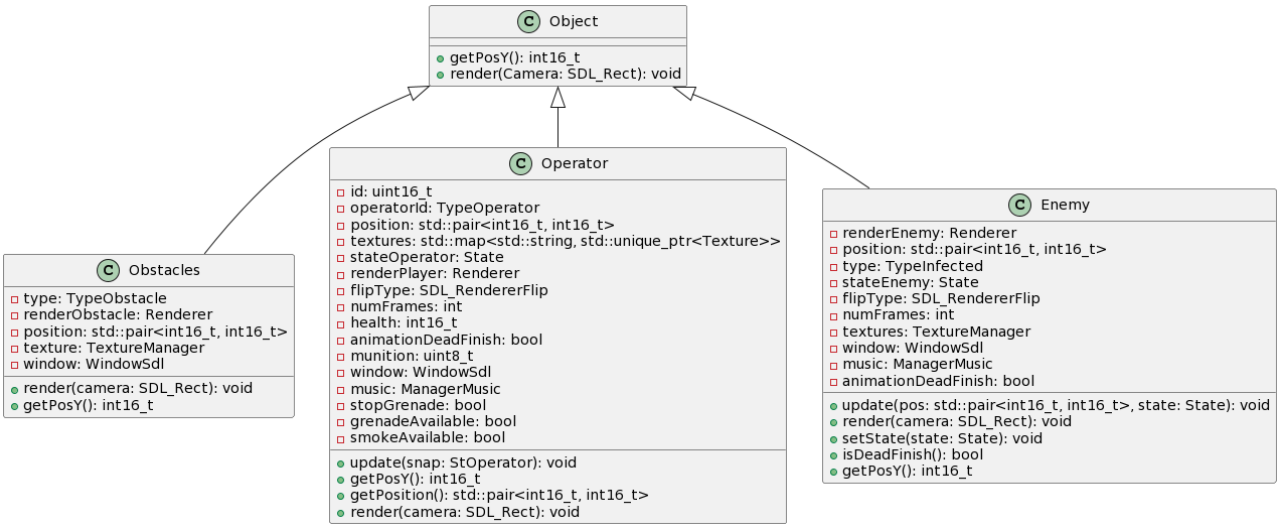

Figura 8: Diagrama de secuencia del Handle Event

Otra cosa a mencionar el renderizado del juego es el Hud, el cuál esta diseñado para escalar dependiendo la resolución de la pantalla en la que se juegue, esto se logra usando el método de proporción, el cuál usa la posición absoluta del objeto a renderizar, esto quiere que es un valor fijo que no depende de la pantalla, la cual transforma la posición (x, y), alto y ancho de la imagen desde una resolución inicial a una resolución mayor o menor. Dicho método se encuentra en la clase Window, la cual es usada por el Hud. Para este escalado en particular se uso una resolución inicial de 1440px de ancho.

```
void WindowSdl::adjustedRect(SDL_Rect& rect) {
    int width = getWidth();
    float aspectRatio =
        static_cast<float>(rect.w) / static_cast<float>(rect.h);
    // Resolucion WIDTH_SCREEN_INIT a la que fue estandarizado
    // para escalar
    float scaleFactorX =
        static_cast<float>(width) / static_cast<float>(
                                        WIDTH_SCREEN_INIT);
    int newWidth = static_cast<int>(rect.w * scaleFactorX);
    int newHeight = static_cast<int>(newWidth / aspectRatio);

    float positionRatio =
        static_cast<float>(rect.x) / static_cast<float>(rect.y);
    int newX = static_cast<int>(rect.x * scaleFactorX);
    int newY = static_cast<int>(newX / positionRatio);
```

```
    rect.x = newX;
    rect.y = newY;
    rect.w = newWidth;
    rect.h = newHeight;
}
```

Código 4: Escalado del Hud

5.5. EndGame

El EndGame es la etapa final del juego, esta es única para cada modo de juego. Se activa al recibir el Event::event_end desde el Servidor. Una vez que eso sucede se dejará de actualizar los demás elementos del juego, para mostrar un mensaje en pantalla.

5.5.1. Modo Clear Zone

Este EndGame se basa solo en mostrar un mensaje de fin de juego, y detectar, el evento que posteriormente hará cerrar el juego para volver al Launcher de inicio.

5.5.2. Modo Survival

Este EndGame además de mostrar un mensaje tiene que recibir en otro evento posterior, las estadísticas del jugador. Estas estadísticas muestran información de la partida actual, asi como también su Ranking general en el servidor. Como el anterior EndGame este también detectará el evento de cierre para volver al Launcher.

5.6. Diagramas

5.6.1. Launcher

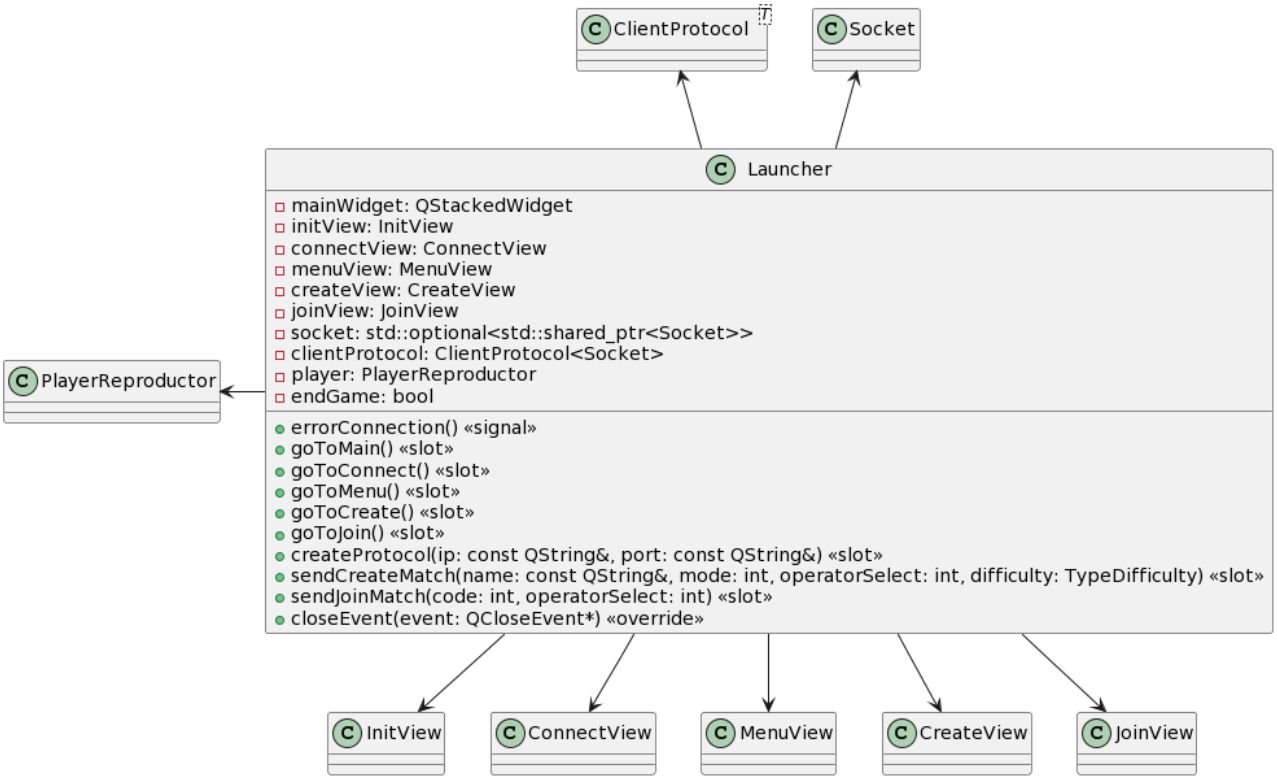

Figura 9: Diagrama de general de clase: Launcher

5.7. Lobby

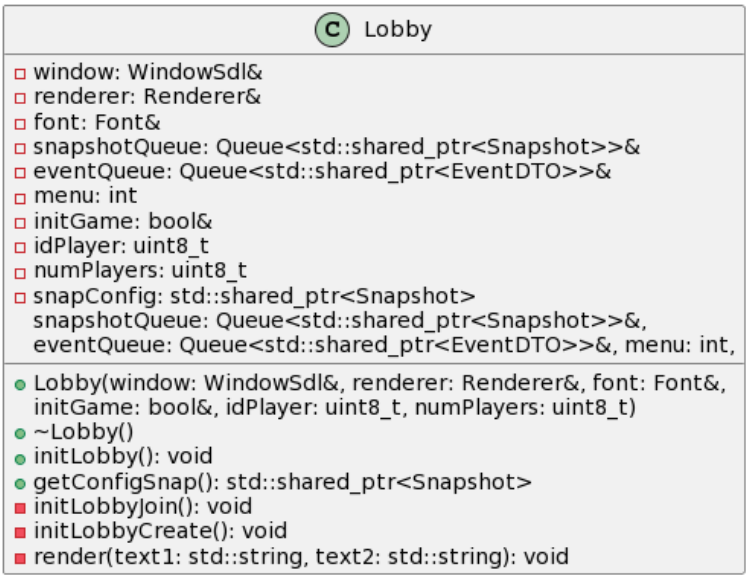

Figura 10: Diagrama de clase de Lobby

5.8. GameSdl

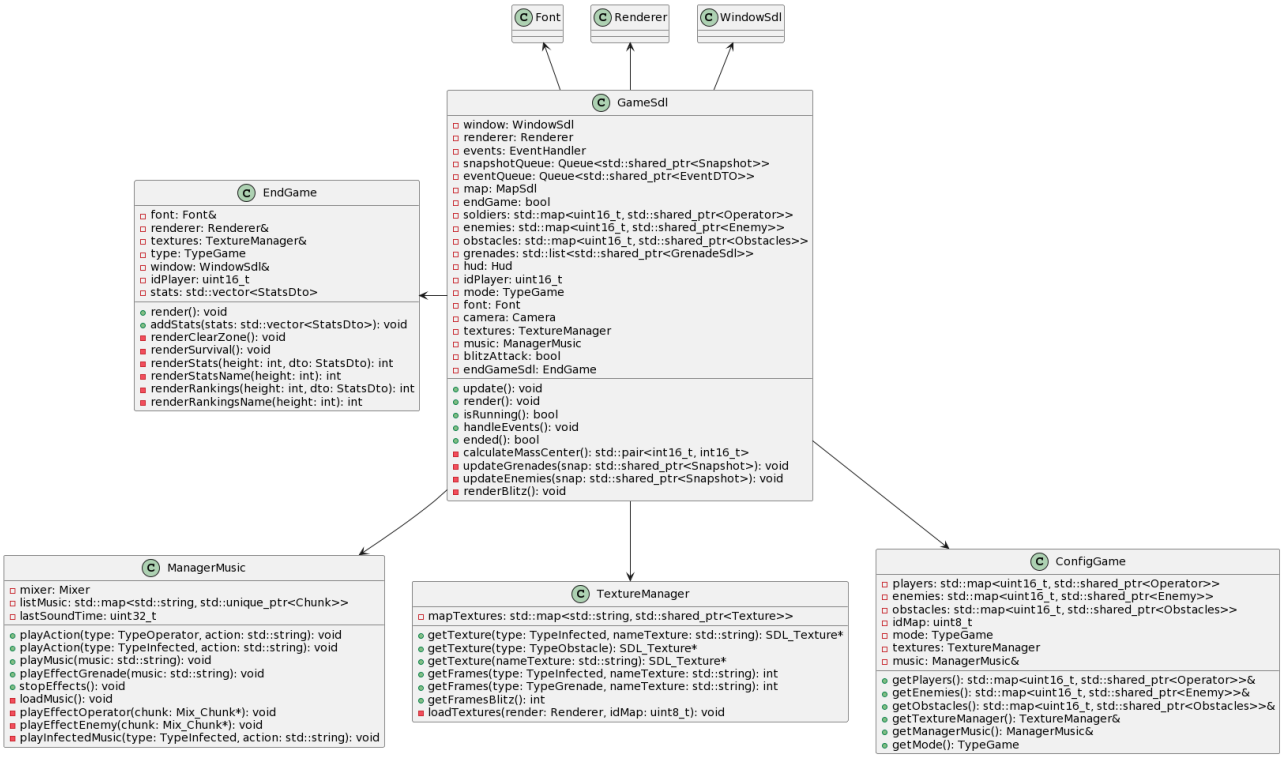

Figura 11: Diagrama de general de clase: GameSdl - parte 1

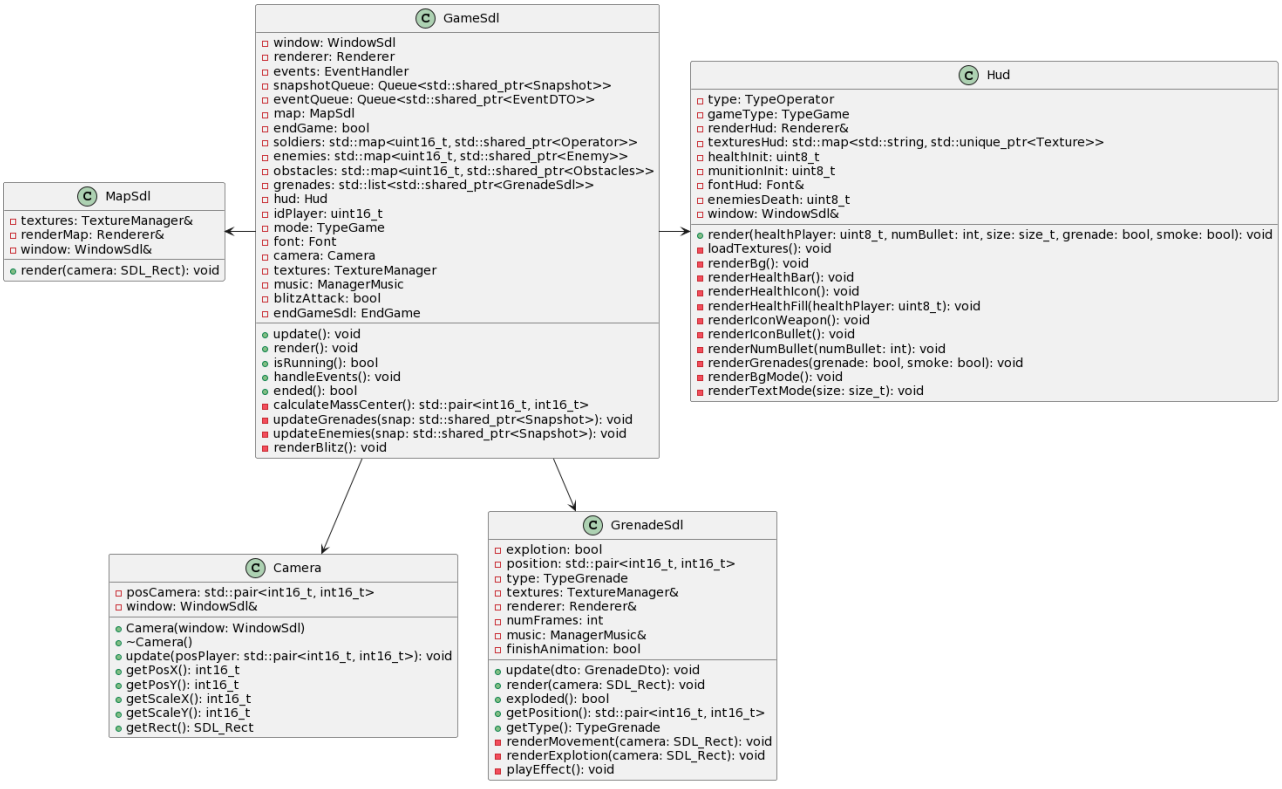

Figura 12: Diagrama de general de clase: GameSdl - parte 2

6. Módulo Servidor

6.1. Descripción General

El servidor es una aplicación distribuida y concurrente, tiene la responsabilidad de recibir conexiones de nuevos clientes y gestionar la lógica de todas las partidas en curso, es decir, ejecuta el bucle que representa el funcionamiento del Left4Dead para cada partida.

6.2. Clases

- **Server:**

  El hilo principal de ejecución se encuentra en la clase Server. Al llamarse al método run() se lanza un hilo **Acceptor** y mientras el mismo se ejecuta, se bloquea leyendo caracteres de 'STDIN' hasta que llega una 'q', en tal caso se fuerza la interrupción del hilo aceptador de clientes y finaliza el programa ordenadamente.

- **Acceptor:**

  Hereda de Thread. Loopea aceptando a los nuevos clientes que se conectan al servidor a través su Socket aceptador. A su vez, el Acceptor es quien tiene el ownership del monitor de partidas **GamesController** (GC). Para cada cliente aceptado lanza un hilo **EventReceiver** pasandole la referencia al GC. Al cerrar el server, se encarga de eliminar los hilos que ya finalizaron.

- **GamesController:**

  Monitor sobre el map que guarda las instancias de **Game** y el contador para crear un nuevo Game. Es el encargado de crear un nuevo game y para hacerlo protege el contador para que otro hilo no cree el mismo Game y mantiene el lock hasta que se haya modificado el map de Games. A su vez, es el encargado de intentar unir un cliente a un Game, y para hacerlo debe proteger el map de Games para que no sea modificado mientras se lo esta iterando.

- **EventReceiver:**

  Hereda de Thread. Representa una conexión aceptada por el servidor. Tiene el ownership de la SnapshotQueue, y del SnapshotSender, este ultimo recibe por referencia el skt y la SnapshotQueue. Utiliza el ServerProtocol para recibir eventos por parte del cliente. Su ejecucion se divide en tres etapas:

  - Etapa inicial:

    En esta etapa se espera la creación de una partida, la unión a una partida, o un leave por parte del cliente. Si se da una creación entonces se le pide al GC que cree el Game, obteniendo así la EventQueue de la partida. Si se da una unión, se le pide al GC que intente unir al cliente a la partida. Si se da un leave se cierra la SnapshotQueue y finaliza la conexión.

  - Etapa Lobby:

    En esta etapa se espera a que llegue el evento de start del Game, o bien que llegue un leave por parte del cliente.

  - Etapa Playing:

Una vez que comenzó la partida, el EventReceiver recibe eventos por parte del cliente y los pushea a la EventQueue del Game.

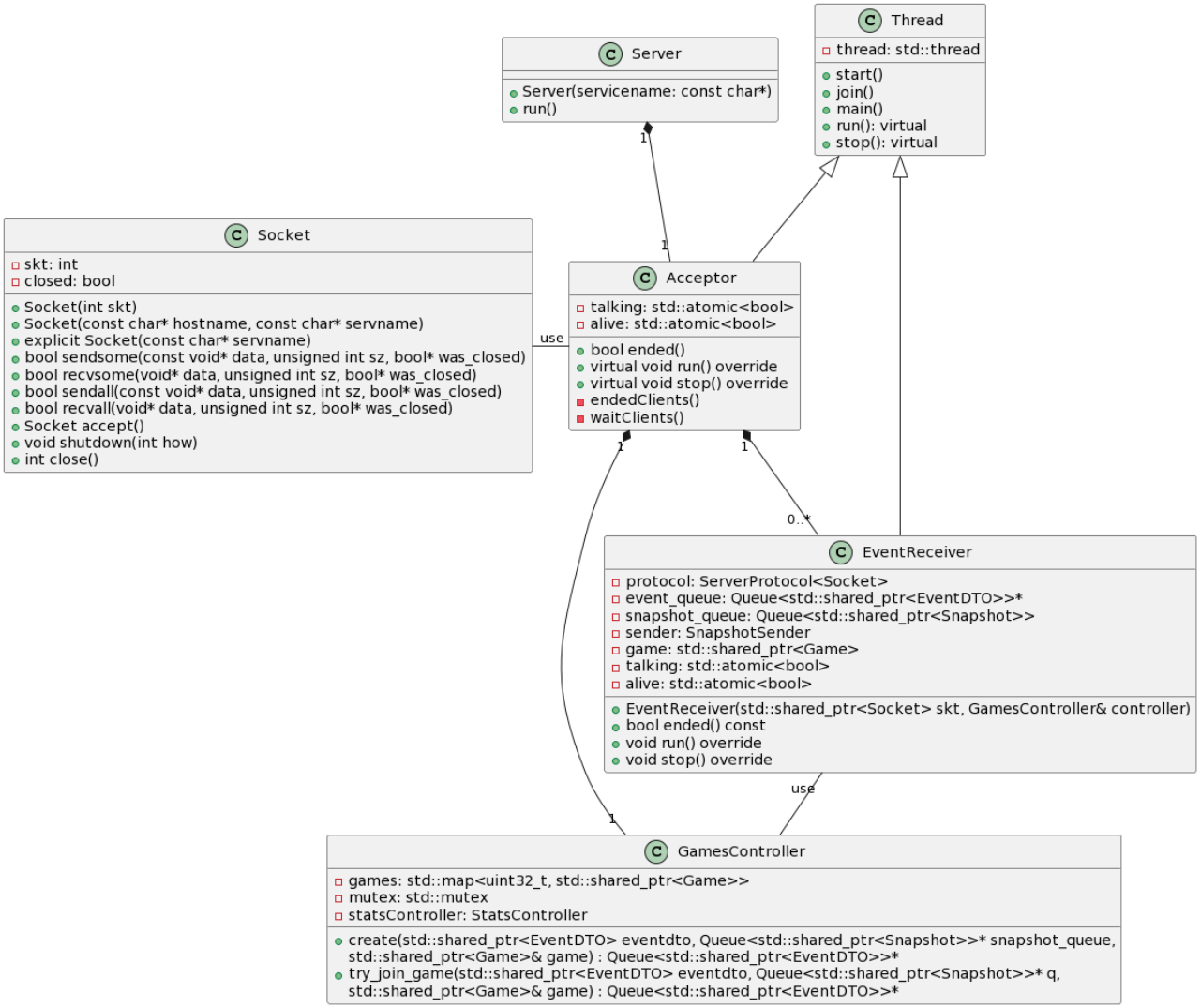

Figura 13: Diagrama de clase general: Servidor

- **Game**:

  Hereda de Thread. Es creado por el monitor de partidas, es la entidad que contiene las SnapshotQueue de cada cliente unido. A su vez tiene el ownership de la EventQueue que contiene los eventos no procesados, a la que cada EventReceiver pushea eventos de cliente. Ejecuta el gameLoop, que en cada iteración procesa eventos que obtiene haciendo pops no bloqueates de la EventQueue. Cada evento es mapeado a un **Command** utilizando la **CommandFactory**, y al llamar a su metodo execute() saben que accion especifica deber realizar sobre el GameWorld, estas acciones setean un estado en particular, y la ejecución efectiva de todos los eventos se da cuando se simula el step de tiempo. Luego de simular un step se le pide al GameWorld el Snapshot que contiene el estado actual del juego, entonces ese Snapshot es broadcasteado a todos los clientes conectados, es decir se pushea el puntero compartido Snapshot a la SnapshotQueue de cada cliente.

- **SnapshotSender:**

  Hereda de Thread. Es el encargado de desencolar los Snapshots de la SnapshotQueue del cliente y enviarlos al servidor mediante el protocolo.

- **Command:**

  Interfaz a la cual se le implementa el patrón Command. Cada hija representa una acción que puede realizar el jugador, ya sea moverse en una dirección, recargar su arma, disparar, lanzar una granada, activar un cheat, abandonar la partida, etc. El metodo execute recibe por referencia el GameWorld, cada hija sabe que acción debe realizar sobre el GameWorld, estas acciones en realidad lo que hacen es setear un estado que posteriormente influirá en la simulación de los steps del gameLoop.

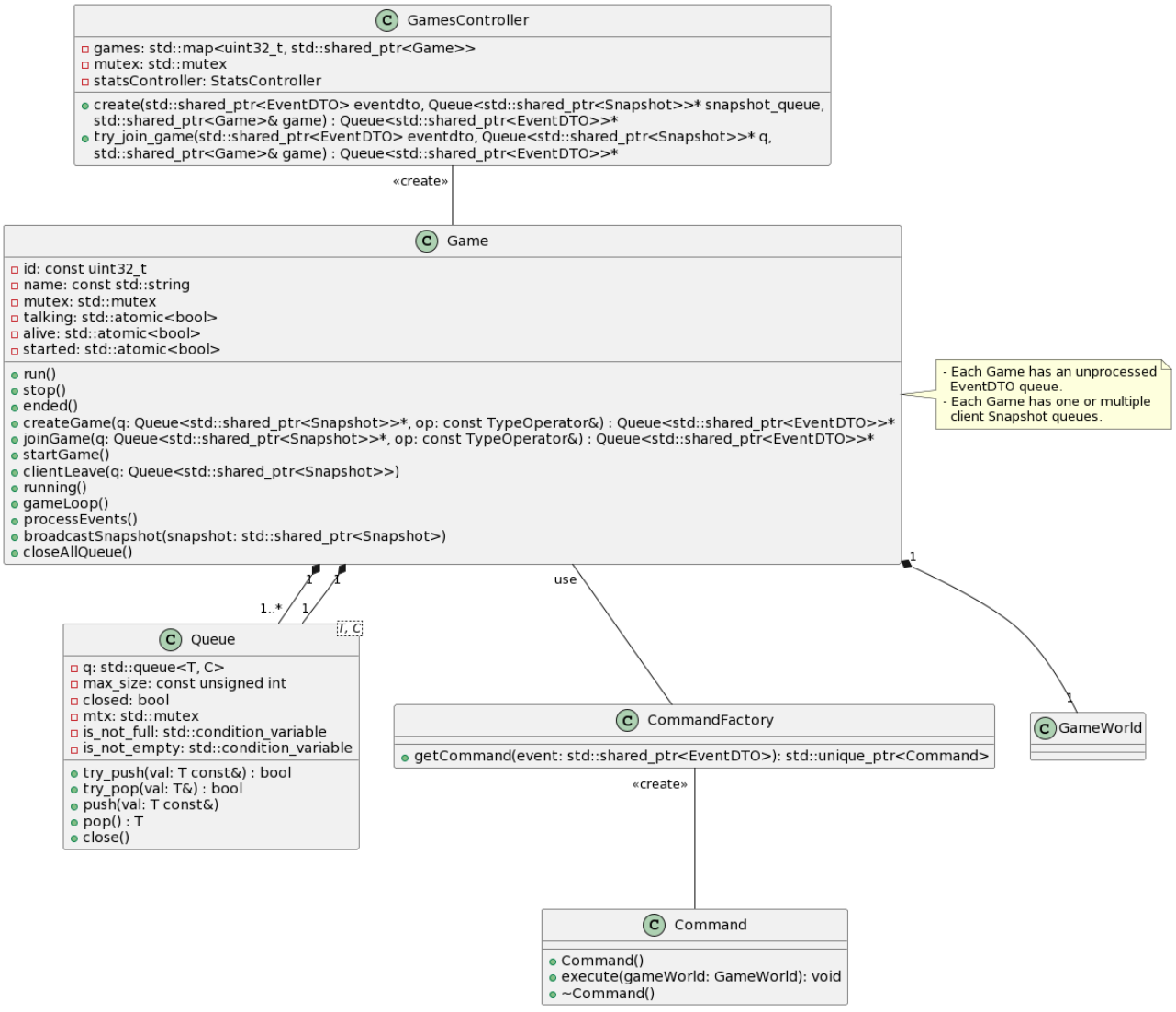

Figura 14: Diagrama de clase: Game

- **GameWorld**

  Funciona como un estilo de patrón Facade, ya que conoce qué clases del subsistema son res-

ponsables de una determinada petición, y delega esas peticiones de los clientes a los objetos apropiados del subsistema. Dichas peticiones son realizadas por los ya mencionados Commands para setear por ejemplo el estado del Player, o bien por parte del Game para aplicar un step de tiempo en el GameWorld. GameWorld contiene al conjunto de todos los Players, Infecteds, Grenades, Obstacles y BlitzAttacks vivos o activos, en los delega casi la totalidad de la lógica del juego, encargándose solamente manejar la lógica del transcurso de rondas, la finalización del juego ya sea victoria o derrota, la generación de obstáculos e infectados, la generación aleatoria del tipo de mapa. Por lo que simular un step en el GameWorld consiste en aplicar un step en cada uno de los objetos mencionados anteriormente. Por otro lado, GameWorld se encarga de crear los Snapshot que tienen toda la información de la partida en curso, así como también de generar las estadísticas una vez finalizado el juego.

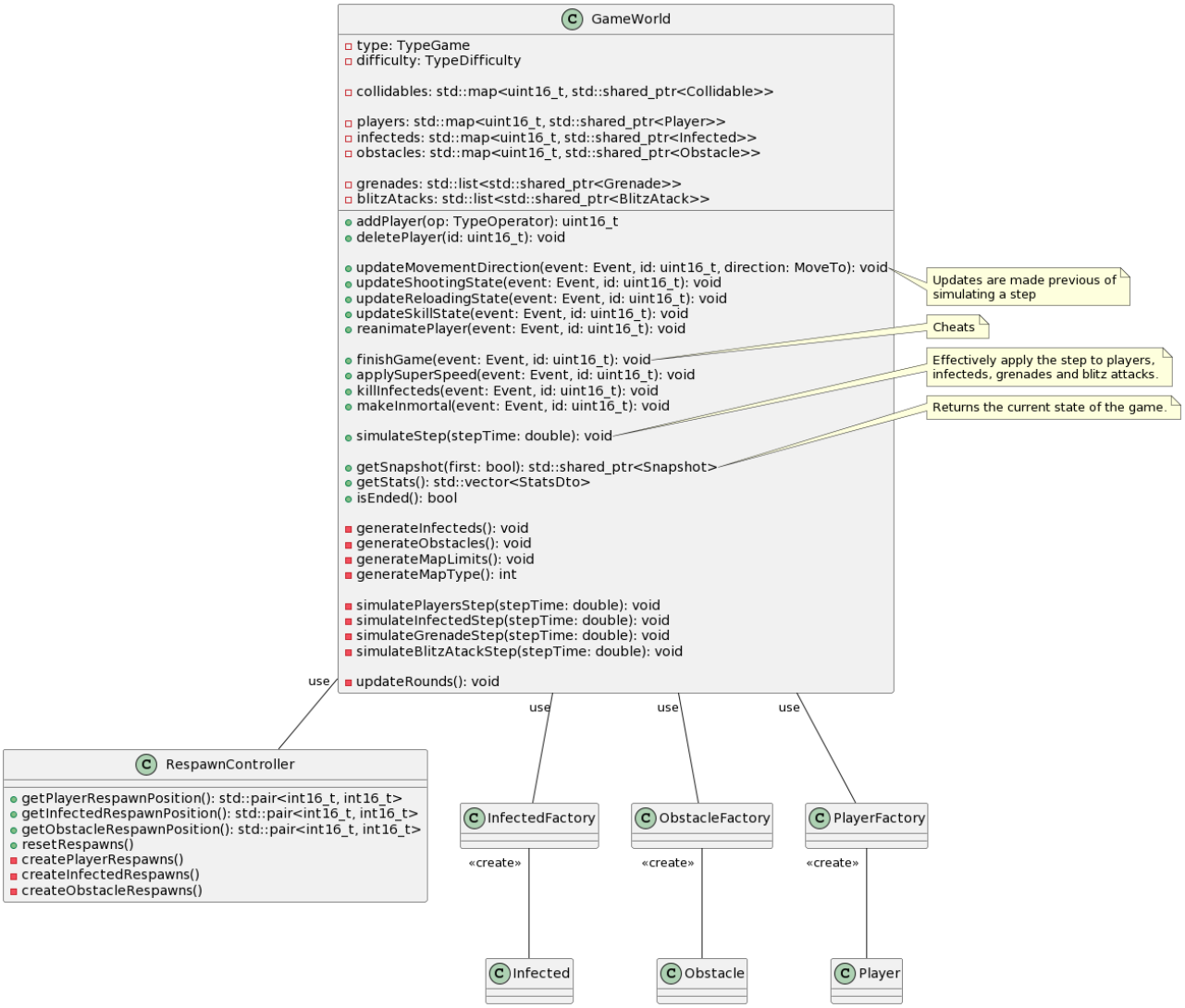

Figura 15: Diagrama de clase: GameWorld

- **Collidable**

La clase Collidable encapsula funcionalidades relacionadas con colisiones, como detección de colisiones, actualizaciones de posición, verificaciones de alineación y cálculos de distancia. Es

una clase fundamental para el desarrollo e implementación de las funcionalidades básicas del juego, ya sea para ejecutar un movimiento, disparos, ataques especiales, etc.

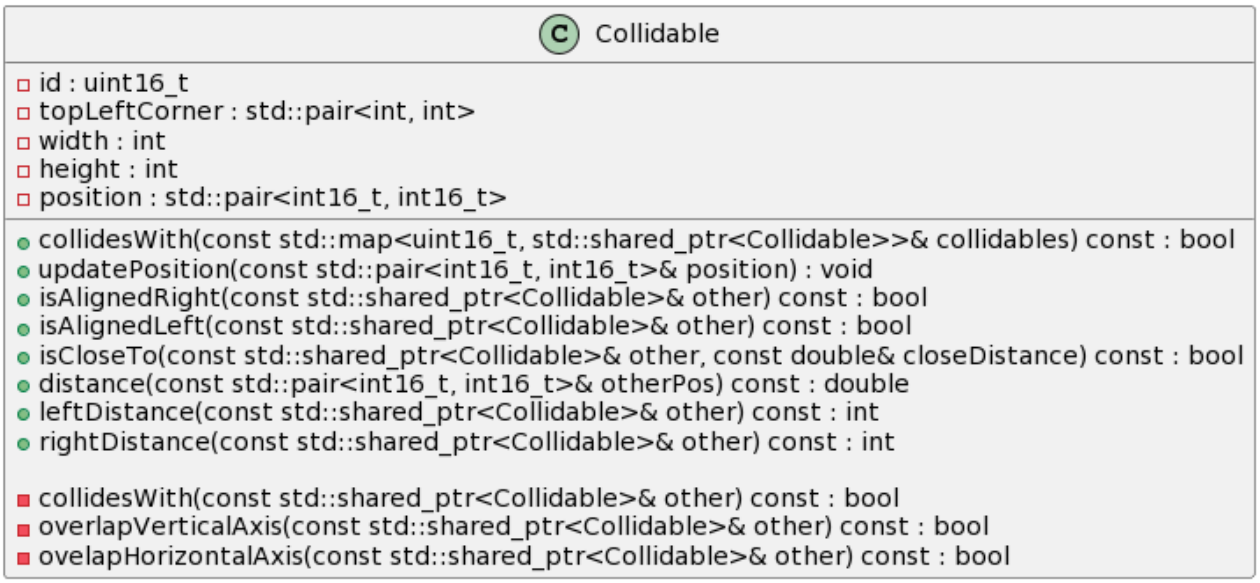

Figura 16: Diagrama de clase: Collidable

- **Player**

  Esta clase es la representación de un cliente dentro del GameWorld, quien delega en Player la responsabilidad de llevar a cabo todas las acciones realizadas por el cliente, en una primera instancia para setear el estado del player, y posteriormente para aplicar un paso de tiempo. Dado que los jugadores podrán elegir entre 3 tipos de soldados que se distinguen por las armas que llevan y también por la habilidad especial, se implemento este punto haciendo uso de herencia y polimorfismo. Los tipos de player son:

  - **P90Player**

    No puede lanzar granadas pero puede iniciar un bombardeo aéreo.

  - **IDFPlayer**

    Puede lanzar granadas tanto explosivas como de humo. Su Weapon es un rifle de asalto IDF.

  - **SCOUTPlayer**

    Puede lanzar granadas tanto explosivas como de humo. Su Weapon es un rifle Scout.

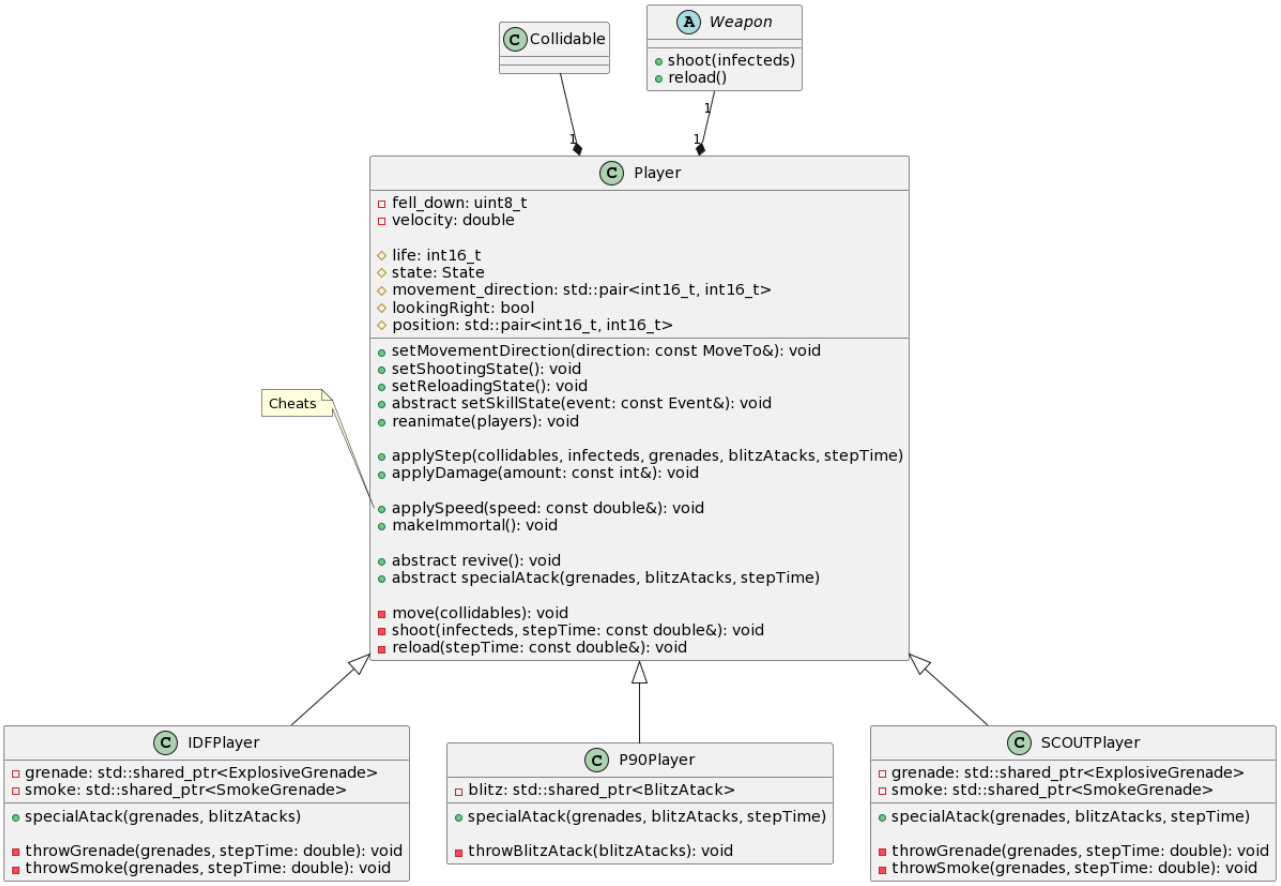

Figura 17: Diagrama de clase: Player

- **Weapon**

  El Player delega en su Weapon la responsabilidad de realizar los disparos, y recargar.

  - **IDF**

    Rifle de asalto que produce una rafaga de 20 balas con un daño considerable a corta distancia pero uno mucho menor a larga distancia reflejando su imprecisión. Debe recargar el arma cada 50 ráfagas.

  - **P90**

    Con ráfagas de 10 balas, es un rifle más balanceado con una performance más consistente en largas distancias. Debe recargar el arma cada 30 ráfagas.

  - **SCOUT**

    Es un rifle que dispara de a 1 bala. No obstante cada bala daña a todo infectado que se encuentre en su trayectoria, el daño de cada impacto se verá reducido conforme la bala pasa por más infectados. Se recarga cada 20 balas.

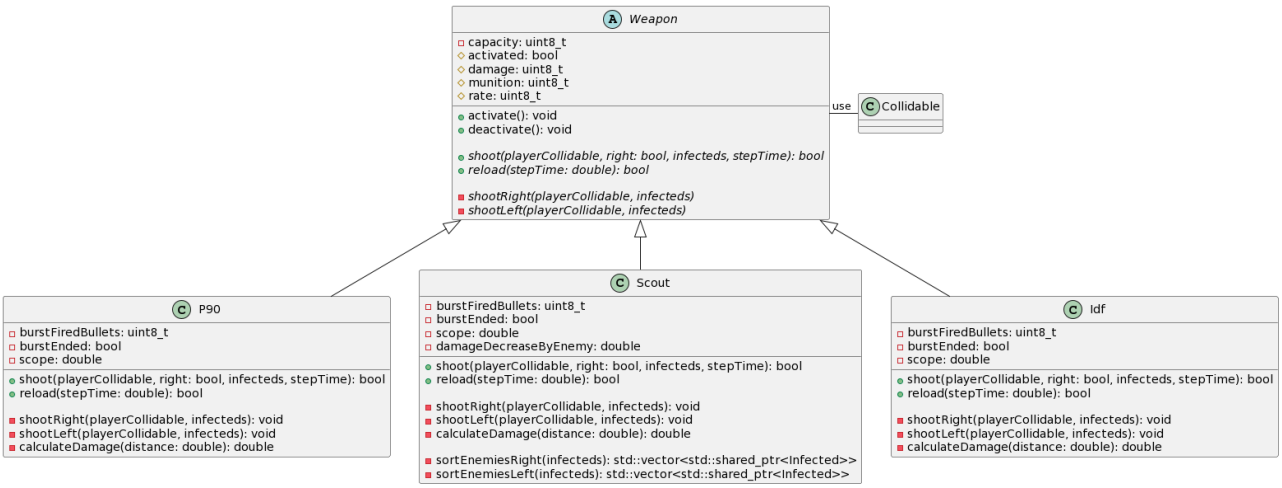

Figura 18: Diagrama de clase: Weapon

- **Habilidades especiales:**

  El player delega en sus habilidades especiales la responsabilidad de lanzarse en determinada dirección en el caso de las granadas, o posición en el caso del ataque aéreo.

  - **Grenade**

    SCOUTPlayer e IDFPlayer pueden lanzar granadas tanto explosivas como de humo. Una vez lanzadas, las granadas tienen un tiempo de recarga que el operador deberá esperar para poder volver a utilizar la habilidad.

    - **ExplosiveGrenade**

      Producen una explosión q daña a todos los infectados y jugadores que se encuentren en el rango de impacto.

    - **SmokeGrenade**

      Las granadas de humo realentizan a los infectados que se encuentran en el rango de impacto al momento de la explosión.

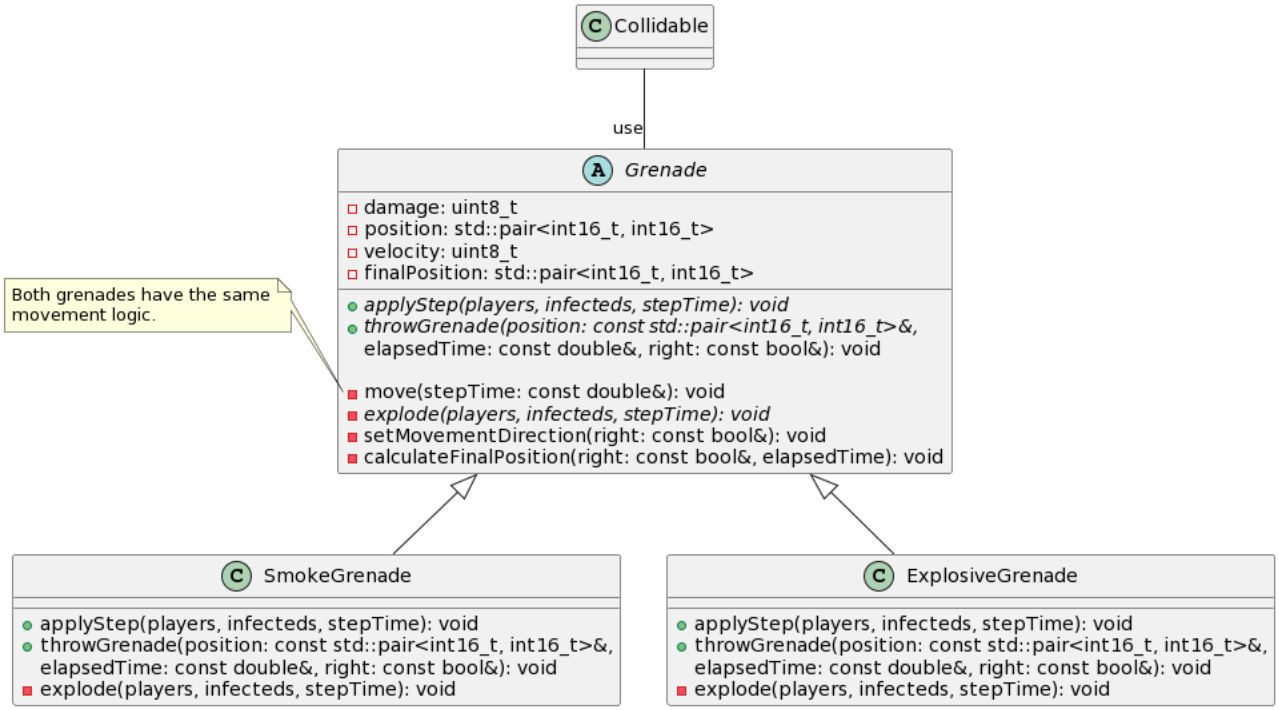

Figura 19: Diagrama de clase: Grenade

- **BlitzAttack**

  El jugador con la P90 no puede lanzar granadas pero puede iniciar un bombardeo aéreo. El bombardeo dejará caer granadas por todo el escenario dañando únicamente a los infectados.

Figura 20: Diagrama de clase: BlitzAttack

- **Infected**

  Los infectados caminan o permanecen quietos en las calles si no son molestados pero la mayoría empezará a caminar hacia los players en cuanto estén a la vista para matarlos. Además de este comportamiento compartido, hay diferentes tipos de infectados, cada uno con cualidades y ataques que los hacen únicos. Por tal motivo se utilizo herencia y polimorfismo. Los tipos de infectados son:

  - **Witch**
  - **Spear**

- **Venom**
- **Zombie**
- **Jumper**

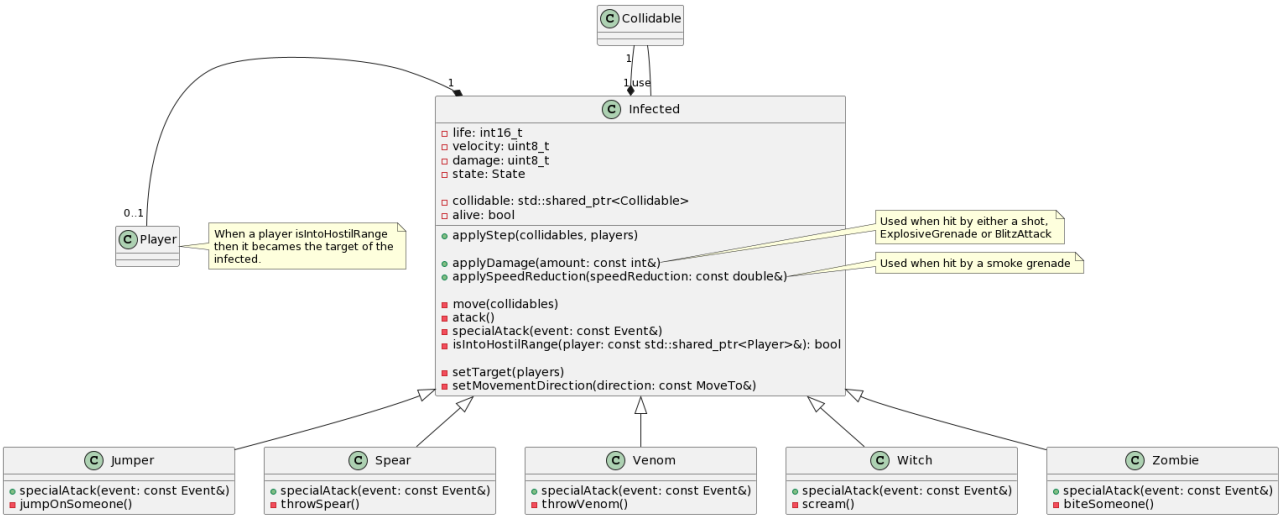

Figura 21: Diagrama de clase: Infected

- **Obstacle**

  Los obstáculos se generan aleatoriamente por todo el mapa, a su vez el tipo de obstáculo también es aleatorio. Para este caso también se utilizo herencia y polimorfismo debido a que los obstáculos difieren en su tipo y dimensiones. Los tipos de obstáculo son:

  - **Crater**
  - **Tire**

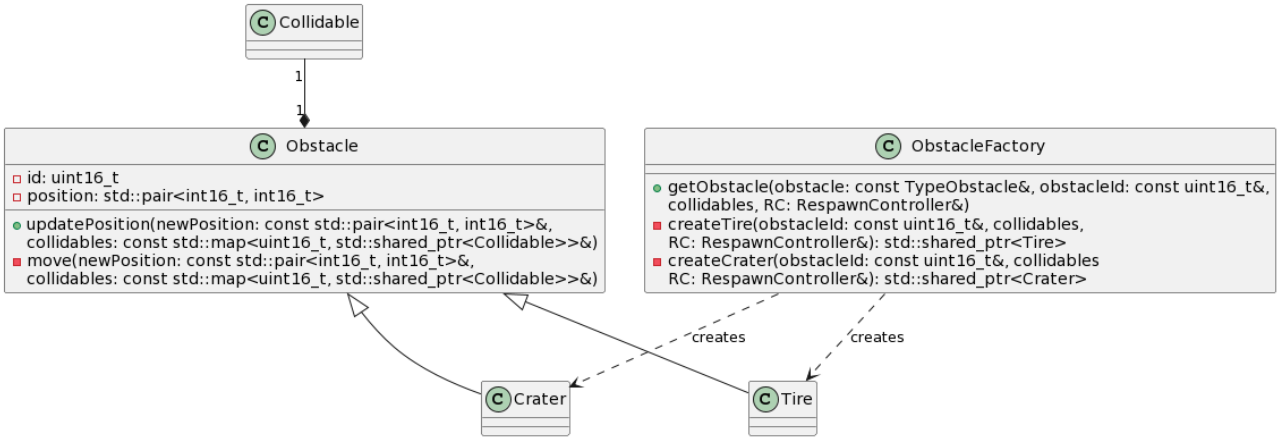

Figura 22: Diagrama de clase: Obstacle