Documentación técnica

Índice

1. Requerimientos de software

Para correr el proyecto, es necesario utilizar como sistema operativo Linux, y las herramientas necesarias para compilar, desarrollar y probar el programa son las siguientes:

- Box2D versión 2.1.2
- Qt versión 5
- QtMultimedia versión 5
- Yaml-cpp versión 0.6
- g++ versión 5.4 o superior
- Cmake versión 3.1 o superior

2. Descripción general

El proyecto se descompone en los siguientes modulos generales:

- Servidor
- Cliente
- Editor
- Common
- Resources
- Config

Los mismos serán explicados en detalle a continuación junto con sus respectivos diagramas y archivos.

3. Modulos

El proyecto contiene los siguientes paquetes:

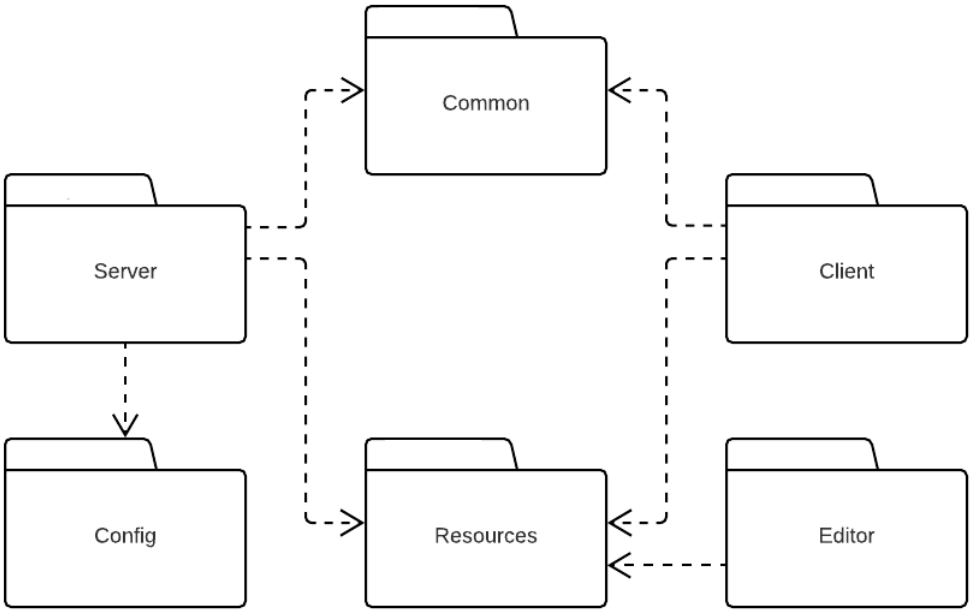

Figura 1: Paquetes generales del juego y sus relaciones

Los mismos serán detallados a continuación:

- **Server**: Contiene la lógica para la creación de un servidor (Es un ejecutable del proyecto).
- **Client**: Contiene la lógica para crear la interfaz gráfica del cliente (Es un ejecutable del proyecto).
- **Editor**: Contiene la lógica para crear un editor de mapas (Es un ejecutable del proyecto).
- **Config**: Contiene los archivos de configuración del juego.
- **Resources**: Contiene todos los recursos utilizados por el editor de mapas, el servidor y el cliente, más específicamente, las imagenes, los sonidos y los mapas a utilizar.
- **Common**: Contiene todos los archivos en común entre el cliente y el servidor.

3.1. Servidor

3.1.1. Descripción general

El servidor es el encargado de aceptar conexiones con jugadores, recibir mensajes y comunicarle mensajes a los mismos, asi como también manejar salas (creación y unión de jugadores a las mismas) y la creación de juegos con sus respectivos jugadores. El mismo se ejecuta permanentemente esperando por entrada estándar el caracter Q para finalizar su ejecución (Bloquear nuevos clientes y esperar a que terminen todos los juegos). A su vez, procesa todas las físicas del juego y la lógica del mismo.

3.1.2. Organización

El servidor consiste en los siguientes paquetes:

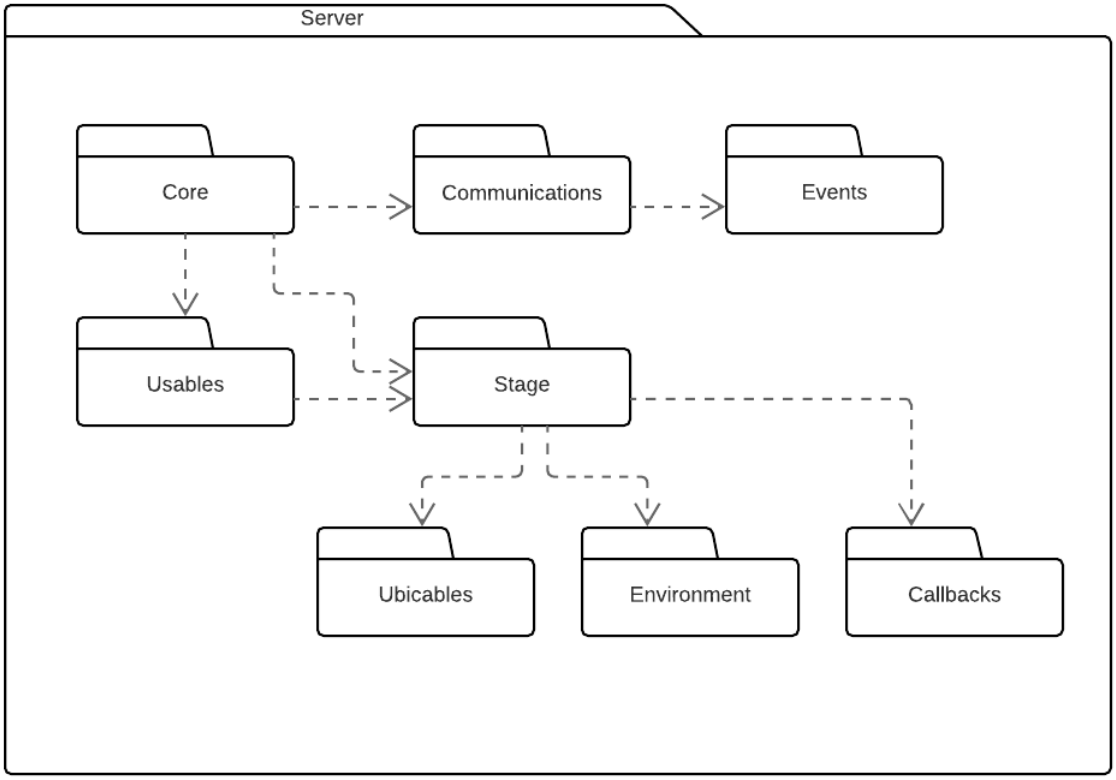

Figura 2: Paquetes del servidor y sus relaciones

Donde cada paquete se detallará a continuacion:

- **Core**: Contiene las bases de un juego, un servidor y jugadores que se conectan al mismo.
- **Communications**: Se encarga de las comunicaciones vía protocolo con el jugador.
- **Events**: Contiene eventos que ocurren y se deben de comunicar a un jugador. Suceden, para cada cliente, desde la conexión del mismo al servidor hasta que se desconecta.
- **Stage**: Contiene el escenario principal de juego, donde se ubicarán todos los objetos.
- **Usables**: Contiene todas las armas y herramientas disponibles en el juego.
- **Ubicables**: Contiene todos los objetos que se pueden insertar en el escenario.
- **Environment**: Contiene todos los objetos que están presentes en el ambiente del escenario e interactúan con ciertos tipos de objetos (Ej: Agua y viento), alterando sus estados.
- **Callbacks**: Contiene las clases que han de ser implementadas del framework Box2D para atrapar los eventos que ocurren en el escenario.

3.1.3. Clases

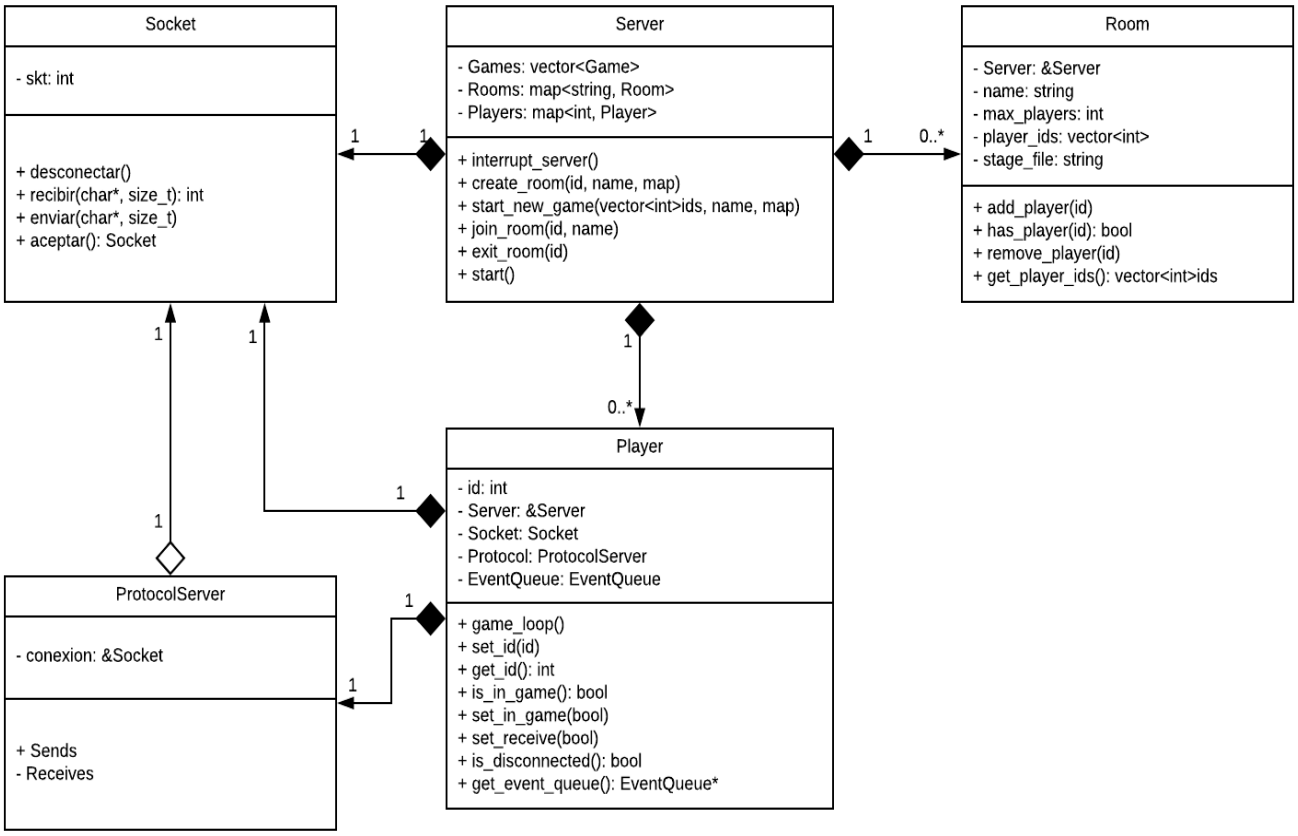

Figura 3: Diagrama de clases general del servidor, salas y jugador

En dicho diagrama se puede ver la relación que existe entre las clases mostradas:
Un servidor que contiene las salas, jugadores que se conectan al mismo y sus respectivos sockets (Socket) para manejar su conexión, así como un protocolo de comunicación del servidor para cada cliente (ProtocolServer)

Nombre	**Server**
Responsabilidad	Se encarga de aceptar las conexiones de los nuevos jugadores. El mismo se puede interrumpir escribiendo el caracter "Q"por entrada estándar.
Atributos	- Games: vector<Game> - Rooms: map<string, Room> - Players: map<int, Player>
Métodos	+ interrupt_server(): Interrumpe el servidor. + create_room(id, name, map): Crea una sala con el nombre y mapa indicado. + start_new_game(vector<int>ids, name, map): Arranca un nuevo juego con los jugadores dados. + join_room(id, name): Une a un jugador a la sala con el nombre dado. + exit_room(id): Expulsa al jugador de la sala en la que se encuentre. + start(): Loop para aceptar nuevos jugadores

Cuadro 1: Clase Server

Nombre	**Room**
Responsabilidad	Contiene los id de los jugadores que están conectados en dicha sala. Se encarga de empezar una partida cuando la misma se encuentra llena.
Atributos	- Server: Server& - name: string - max_players: int - player_ids: vector<int> - stage_file: string
Métodos	+ add_player(id): Agrega al jugador a la sala. + has_player(id): Devuelve si el jugador está en la sala. + remove_player(id): Expulsa al jugador de la sala. + get_player_ids(id): Devuelve todos los jugadores de la sala.

Cuadro 2: Clase Room

Nombre	**Player**
Responsabilidad	Se encarga de manejar los mensajes que se envían y reciben de los clientes, así como también de controlar a los worms que tiene a su disposición.
Atributos	- id: int - server: Server& - socket: Socket - protocol: ProtocolServer - event_queue: EventQueue - worms: map<int, Worm> - usables: map<int, Usable>
Métodos	+ game_loop(): Loop para recibir información del jugador. + get_id(): Devuelve el id del jugador. + is_in_game(): Devuelve si el jugador está jugando. + set_in_game(bool): Setea que el jugador está o no jugando. + set_receive(bool): Setea que el jugador puede o no recibir información a través del protocolo. + is_disconnected(): Devuelve si el jugador está desconectado. + get_event_queue(): Devuelve la cola de eventos del jugador. + play(): Habilita al jugador para que pueda jugar.

Cuadro 3: Clase Player

Nombre	**ProtocolServer**
Responsabilidad	Clase que representa un protocolo de comunicación de datos desde el lado del servidor. Envía los datos desde el mismo al jugador correspondiente.
Atributos	———
Métodos	+ sendWormId(id, worm_id, health): Envía el worm que le pertenecerá al jugador con dicho id. + sendUsableId(id, ammo): Envía las armas que el jugador puede utilizar junto a la cantidad de municiones. + sendPlayerId(id): Envía el id del jugador. ...

Cuadro 4: Clase ProtocolServer

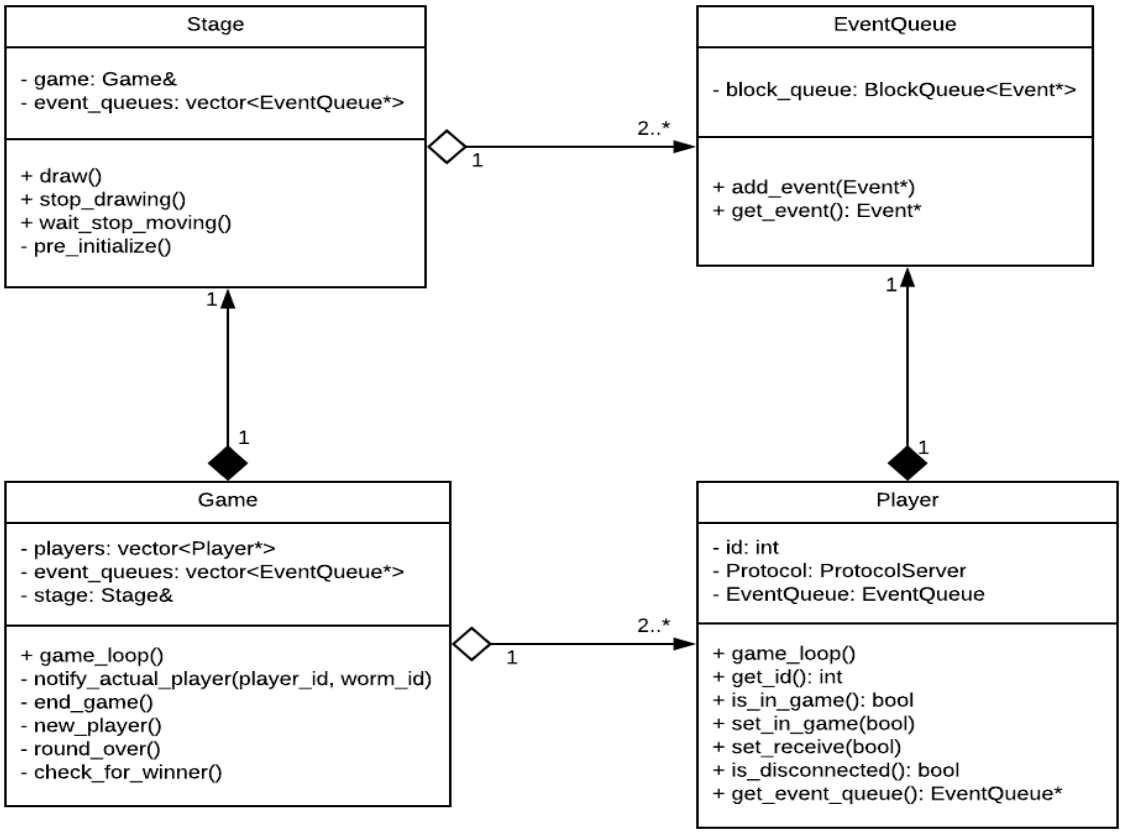

Figura 4: Diagrama de clases de Game y sus relaciones

En este diagrama, se pueden ver las siguientes relaciones:

- Cada juego (Game) contiene un escenario propio donde van a figurar los objetos.
- Cada juego (Game) contiene una referencia a todos los jugadores que van a jugar, ya que el mismo persiste en el servidor hasta que le juego termine y se lo desconecte efectivamente.
- Cada jugador (Player) tiene su propia cola de eventos (EventQueue), de manera que con la misma se puedan ir encolando eventos que el mismo tiene que procesar para cambiar su estado y notificarle al jugador vía protocolo.
- El escenario (Stage) tiene acceso a las colas de eventos de los jugadores, de manera que pueda agregarle eventos a los mismos sobre hechos que ocurran en el escenario (Ej: Posiciones de los objetos)

Nombre	**Game**
Responsabilidad	Se encarga de crear un escenario y manejar la lógica de los turnos de la partida.
Atributos	- players: vector<Player*> - event_queues: vector<EventQueue*> - stage_file: string - stage_t: thread - is_over: bool - stage: Stage
Métodos	- end_game(): Finaliza el juego. - new_player(): Obtiene un nuevo jugador. - round_over(): Finaliza una ronda. - check_for_winner(): Chequea si hay un ganador. - notify_actual_player(player_id, worm_id): Crea un evento para notificar a todos el jugador actual y su worm a usar. - get_actual_player(): Obtiene al jugador actual. + game_loop(): Loop del juego. + game_finished(): Devuelve si el juego finalizó.

Cuadro 5: Clase Game

Nombre	**Event**
Responsabilidad	Son eventos que el jugador procesa para comunicar datos a través del protocolo al cliente que está jugando.
Atributos	———
Métodos	+ process(Player&, Protocol&): Procesa un evento.

Cuadro 6: Clase Event

Nombre	**EventQueue**
Responsabilidad	Es una cola bloqueante que almacena los eventos que posteriormente el jugador va a procesar.
Atributos	- block_queue: BlockQueue<Event*>
Métodos	+ add_event(Event*): Agrega un evento a la cola. + get_event(): Espera y saca el primer evento de la cola.

Cuadro 7: Clase EventQueue

Nombre	**Stage**
Responsabilidad	Se encarga de procesar las físicas del juego incluyendo el movimiento de los objetos, creación y eliminación de los mismos.
Atributos	- world: b2World - game: Game& - event_queues: vector<EventQueue*> - wind: Wind - water: Water - ubicables: vector<Ubicables*> - movables: vector<Movables*> - continue_drawing: bool
Métodos	+ insert(Ubicable*): Inserta un ubicable en el mundo. + insert(Movable*): Inserta un movable en el mundo. + stop_drawing(): Deja de ejecutar el mundo físico. + wait_stop_moving(): Espera que todos los movables del mundo dejen de moverse. - pre_initialize(): Pre-inicializa los objetos en el mundo. - remove_deads(): Elimina los objetos muertos. - create_objects(): Crea los objetos nuevos.

Cuadro 8: Clase Stage

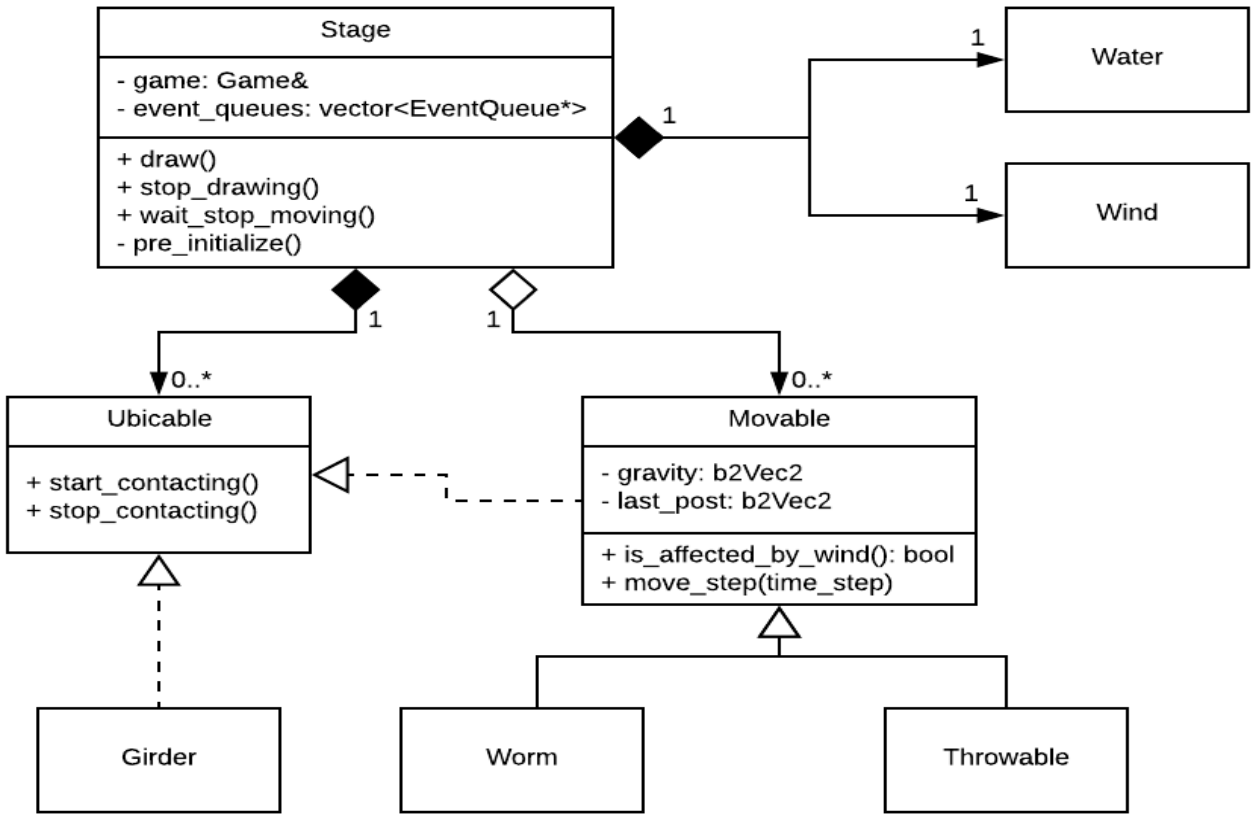

Figura 5: Diagrama de clases de Stage y sus relaciones

El diagrama muestra los objetos que contiene un escenario (Stage), ya sean objetos ubicables estáticos (Ubicable) o móviles (Movable), también, el escenario contiene agua y viento determi-

nados que afectaran dichos objetos ubicables. Particularmente, las vigas (Girder) son ubicables estáticos mientras que los objetos arrojables (Throwable) y gusanos (Worm) son móviles.

Nombre	**Ubicable**
Responsabilidad	Son los objetos que se ubican en el escenario.
Atributos	———
Métodos	+ start_contacting(b2Contact*): Evento para empezar a contactar. + stop_contacting(b2Contact*): Evento para dejar de contactar. + create_myself(b2World&): Crearse en el mundo. + delete_myself(b2World&): Borrarse del mundo. + should_collide_with(Ubicable*): Devuelve si debe de colisionar con otro tipo de ubicable. + im_dead(): Devuelve si está muerto. + get_type(): Devuelve el tipo del ubicable. + get_id(): Devuelve el id del ubicable. + force_death(): Fuerza la muerte del ubicable.

Cuadro 9: Clase Ubicable

Nombre	**Movable**
Responsabilidad	Son ubicables que a su vez se pueden mover en el mundo.
Atributos	- gravity: b2Vec2 - last_post: b2Vec2
Métodos	+ set_position(b2Vec2): Setea la posición del movable. + get_position: Obtiene la posición del movable. + set_gravity(b2Vec2): Setea la gravedad del movable. + is_affected_by_wind(): Devuelve si el movable es afectado por el viento. + move_step(time_step): Evento que se llama en cada ciclo del mundo (Útil para realizar movimientos).

Cuadro 10: Clase Movable

Nombre	**Wind**
Responsabilidad	Representa el viento del mundo.
Atributos	- min_wind_speed: const float - max_wind_speed: const float - wind_speed: float - direction: int
Métodos	+ change_wind(): Cambia el viento aleatoriamente. + apply(Movable*): Aplica el viento al movable. + set_wind_limits(float min, float max): Setea los valores mínimo y máximo posible para el viento. - get_wind_speed(): Devuelve la velocidad del viento.

Cuadro 11: Clase Wind

Nombre	**Water**
Responsabilidad	Representa el agua del juego.
Atributos	- water_level: const float
Métodos	+ set_water_level(float): Setea el nivel del agua. + get_water_level(): Obtiene el nivel del agua

Cuadro 12: Clase Water

Nombre	**Worm**
Responsabilidad	Representa a los gusanos del juego.
Atributos	- game: Game& - actual_health: int - should_slide: bool - sliding: bool - facing_direction: MoveDirection - move_direction: MoveDirection
Métodos	- is_on_ground(): Devuelve si el gusano está en el suelo + start_moving(MoveDirection): Mueve al gusano hacia la dirección indicada. + use(Usable*, b2Vec2 destino, vector<int>params): Usa el arma indicada en la posición destino con los parámetros extra. + add_health(int hp): Agrega vida al gusano. + get_health(): Devuelve la vida del gusano. + is_sliding(): Devuelve si el gusano está deslizando. + receive_dmg(int dmg): Causa el daño indicado al gusano. + receive_explosion(b2Vec2&): Recibe una explosión y es empujado

Cuadro 13: Clase Worm

Nombre	**Throwable**
Responsabilidad	Representa todos los objetos arrojables por las armas/herramientas.
Atributos	- owner: Worm* - velocity: float - radius: float - max_dmg: float - radius_expl: float - max_pushback: float
Métodos	- explode(): Realiza una explosión en el lugar.

Cuadro 14: Clase Throwable

Nombre	**Girder**
Responsabilidad	Representa a las vigas del mundo.
Atributos	———
Métodos	———

Cuadro 15: Clase Girder

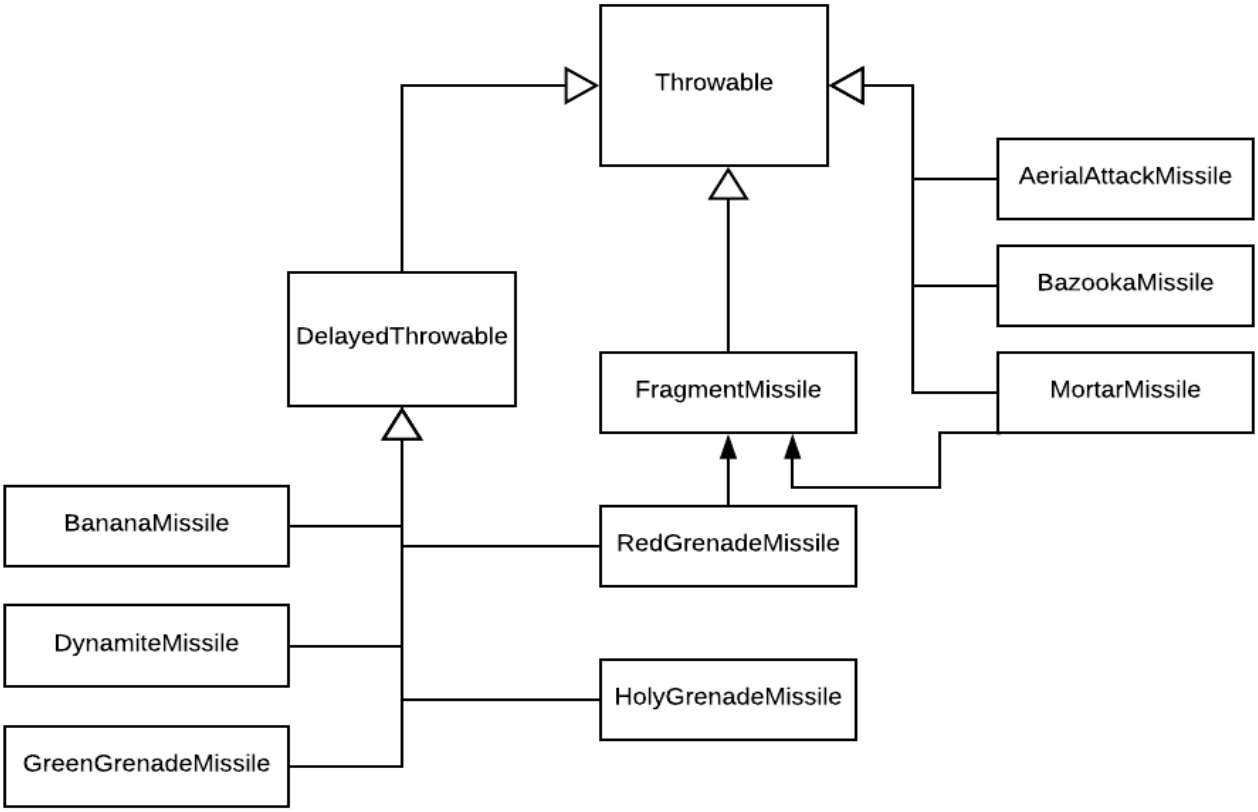

Figura 6: Diagrama de clases de Throwable y sus relaciones

El diagrama muestra los diferentes tipos de objetos arrojables (Throwable) que existen, así como también los arrojables que a su vez tienen un tiempo predefenido antes de explotar (DelayedThrowable).

Nombre	**DelayedThrowable**
Responsabilidad	Es un tipo especial de Throwable que explota después de un tiempo específico.
Atributos	- counter: int
Métodos	+ is_affected_by_wind(): Devuelve si es afectado por el viento.

Cuadro 16: Clase DelayedThrowable

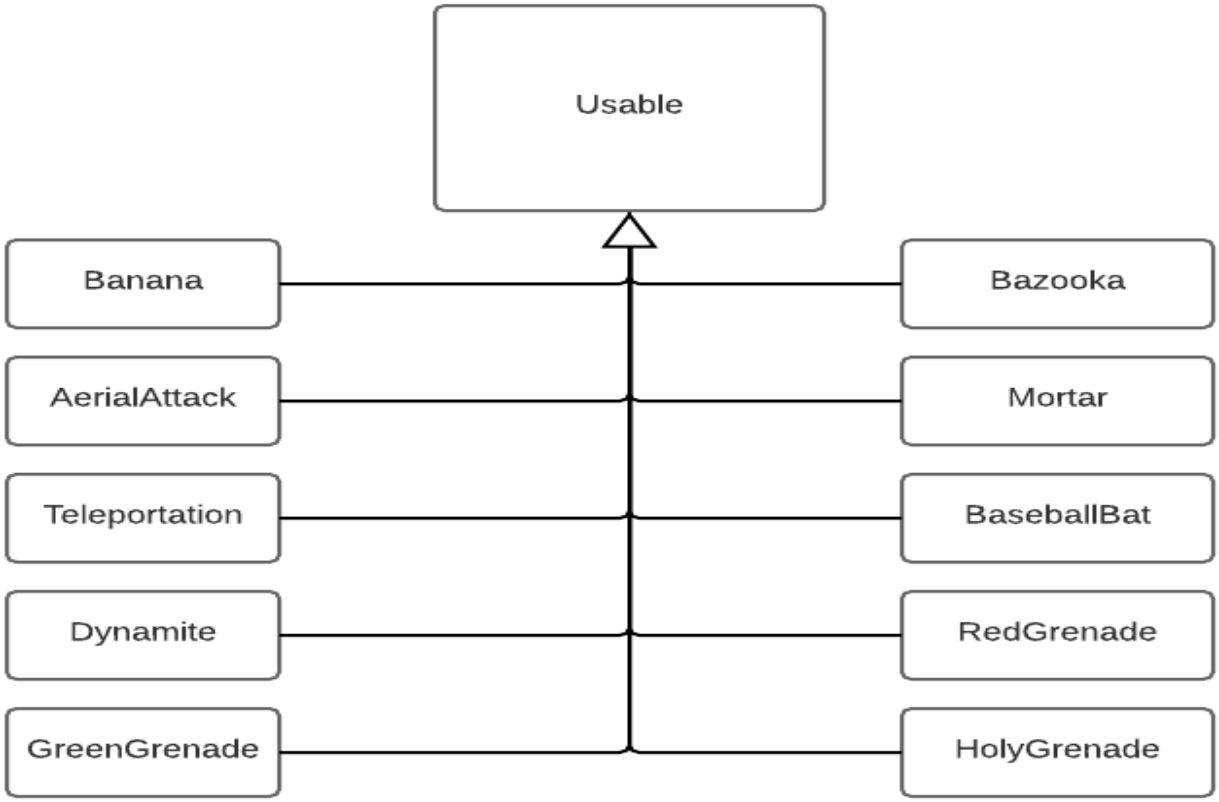

Figura 7: Diagrama de clases de Usable y sus relaciones

El diagrama muestra los diferentes tipos de armas y herramientas (Usable) que existen en el juego.

Nombre	**Usable**
Responsabilidad	Representan las armas y herramientas de los gusanos.
Atributos	- stage: Stage& - ammo: int
Métodos	+ action(Worm*, b2Vec2& destino, vector<int>params): Método a redefinir para especificar que hará el arma. + use(Worm*, b2Vec2& destino, vector<int>params): Utiliza el arma en la posición indicada con los parámetros dados. + get_id(): Devuelve el id del usable + get_ammo(): Devuelve la munición actual del usable.

Cuadro 17: Clase Usable

A continuación, se explicarán las clases que se implementaron del framework Box2D para poder atrapar eventos de colisiones con el mundo físico creado por el mismo.

Nombre	**Explosion**
Responsabilidad	Se encarga de crear una explosión en un cierto radio indicado, causando daño e impulsando a todos los gusanos que se encuentren en ella.
Atributos	———————————————————————————-
Métodos	———————————————————————————-

Cuadro 18: Clase Explosion

Nombre	**Sensor**
Responsabilidad	Es un objeto que forma parte de los gusanos. Se encarga de indicarle al mismo si está deslizando o se encuentra en el suelo para poder realizar un salto o moverse.
Atributos	- worm: Worm& - object_count: int
Métodos	+ add_at_position(b2Body*, b2Vec2 pos, float long, float height): Agrega un sensor en la posición relativa al cuerpo indicada. + get_number_colisions(): Cantidad de ubicables colisionando con el sensor.

Cuadro 19: Clase Sensor

Nombre	**QueryCallback**
Responsabilidad	Clase implementada de Box2D que permite comprobar que objetos se encuentran en una cierta área especificada.
Atributos	+ foundBodies: vector<b2Body*>
Métodos	+ ReportFixture(b2Fixture*): bool

Cuadro 20: Clase QueryCallback

Nombre	**ContactListener**
Responsabilidad	Clase implementada de Box2D que permite atrapar eventos de colisionar y dejar de colisionar para los ubicables.
Atributos	———————————————————————————-
Métodos	+ BeginContact(b2Contact*): Evento de contacto entre ubicables. + EndContact(b2Contact*): Evento dejar de contactar entre ubicables.

Cuadro 21: Clase ContactListener

Nombre	**ContactFilter**
Responsabilidad	Clase implementada de Box2D que permite filtrar que ubicables deberían de colisionar con otros tipos de ubicables.
Atributos	———————————————————————————-
Métodos	+ ShouldCollide(b2Fixture*, b2Fixture*): Establece si dos ubicables deben colisionar.

Cuadro 22: Clase ContactFilter

3.1.4. Diagramas de secuencia

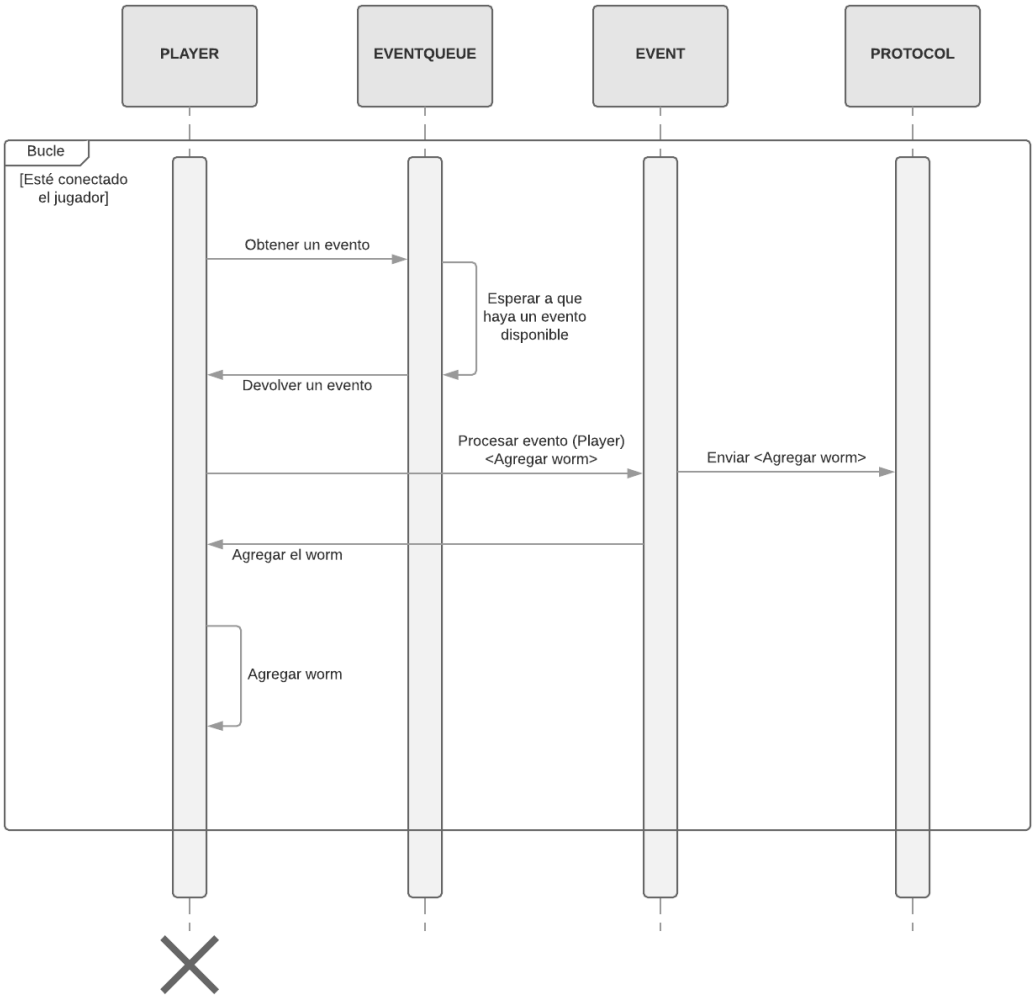

Figura 8: Diagrama de secuencia del procesamiento de eventos

Este diagrama muestra como se procesan los eventos que se le agregan a cada jugador, primero obteniendo los mismos de la cola de eventos (EventQueue), para luego ejecutarlos y enviar la información necesaria al jugador vía protocolo (Protocol).

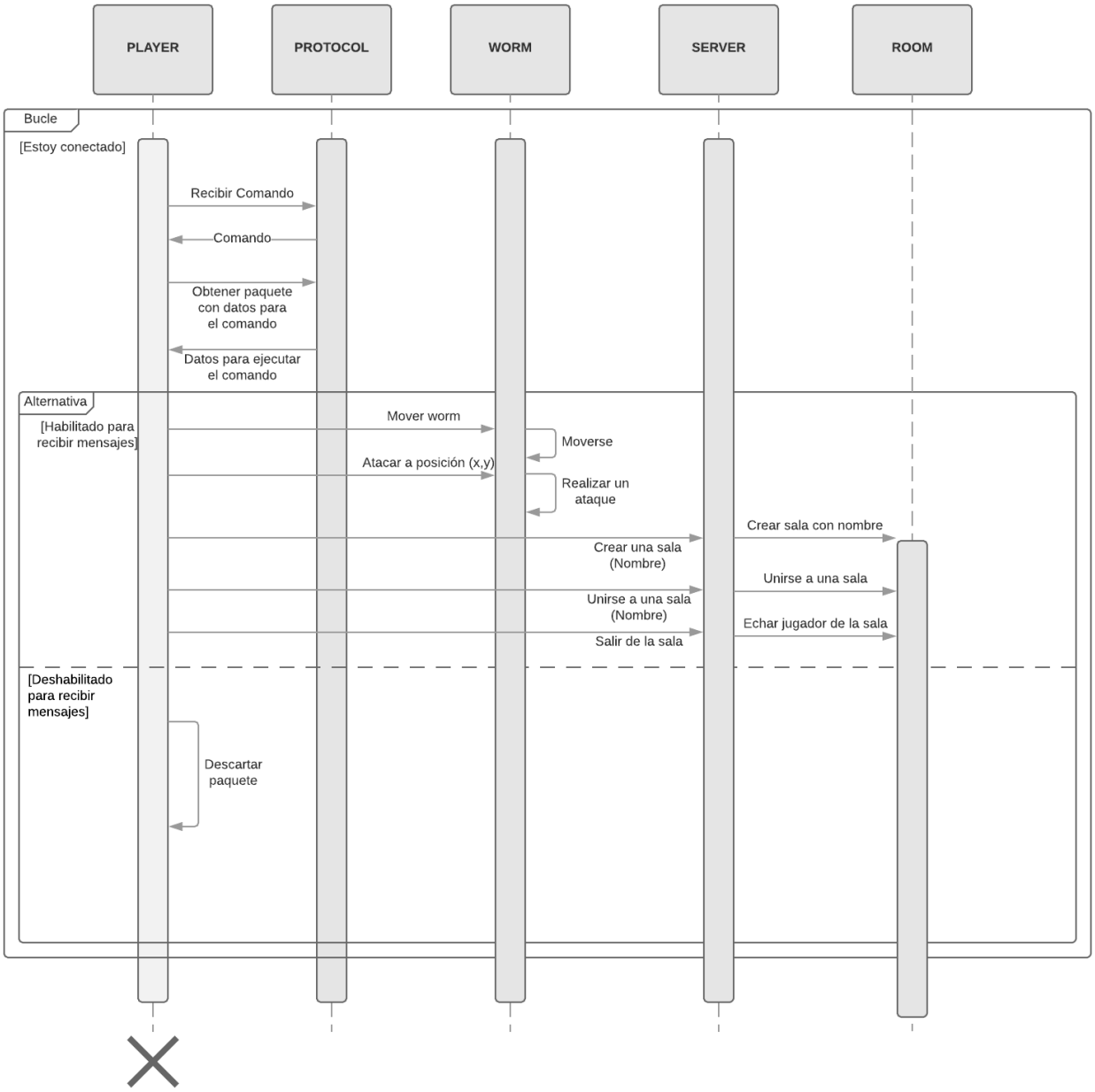

Figura 9: Diagrama de secuencia de recibir información del jugador

Este diagrama muestra como se van procesando los paquetes que envía el jugador al servidor. Se recibe un comando, se obtiene el paquete necesario para dicho comando, y si el jugador tiene habilitado para recibir mensajes, lo procesa, sino, se descarta.

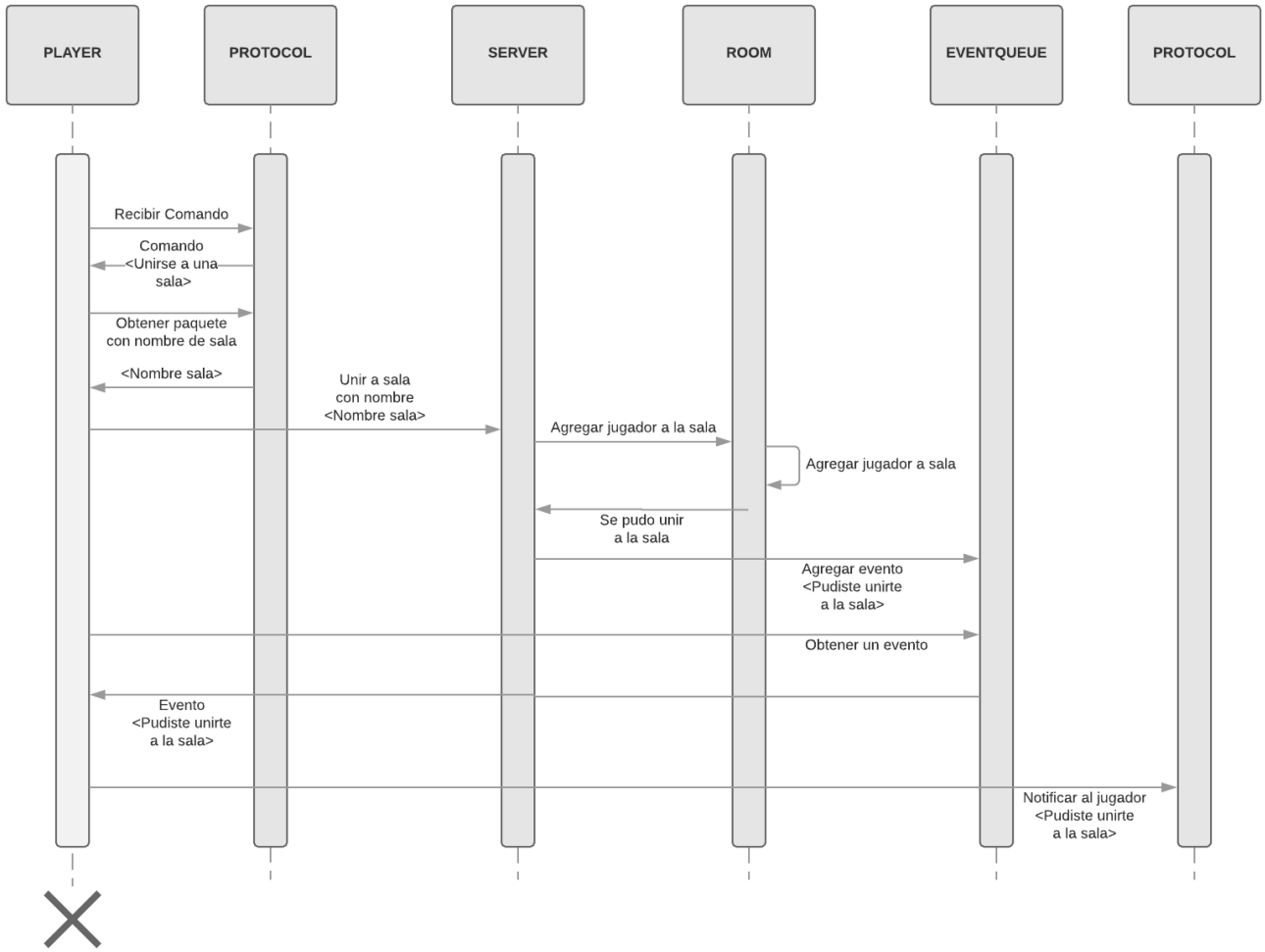

Figura 10: Diagrama de secuencia de unirse a una sala

Este diagrama muestra como el jugador se une a una sala a través de un paquete que le envía el cliente, así como también la respuesta por parte del servidor con un paquete de si se pudo unir a la sala.

3.1.5. Descripción de archivos y protocolos

El servidor maneja archivos de extensión ".yaml", los mismos corresponden a los mapas creados por el editor (Los cuales se encuentran dentro de la carpeta "Resources/maps") y un archivo de configuración que tiene los valores por defecto todos los objetos que se encuentren en el escenario y las entidades que afecten a los mismos, ya sea, por ejemplo, el viento que afecta a los movibles (El mismo se encuentra en la carpeta "Config").
El protocolo que utiliza el servidor es manejado por la clase ProtocolServer anteriormente explicada. Más adelante, se darán en detalle los paquetes que se comunican entre servidor y cliente (Ver sección **Protocolo de comunicación**)

3.2. Editor

El Editor es el encargado de diseñar los mapas para que el servidor lee para hacer el juego. En el momento de ejecucion se muestra una ventana en la cual se da la opcion de crear un nuevo mapa, o cargar uno ya existente.

En el proceso de creacion de mapa se ubican tanto los worms como las girders en las posiciones que se desean. Tambien e deve de selecionar que usables estan disponibles al igual que la municion que tienen durante la duracion del juego.

3.2.1. Organizacion

La siguiente imagen representa la orgenizacion de los distintos paquetes del editor.

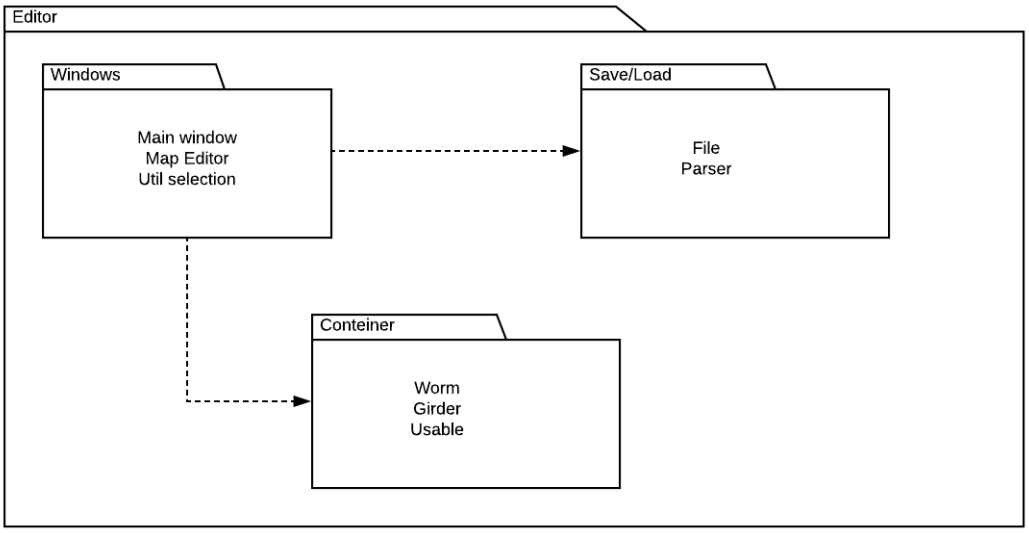

Figura 11: Paquete del Editor y sus relaciones

- Save/Load: contiene la para guardar y cargar el mapa usando la libreria YAML cpp
- Conteiner: En este modulo están las clases en donde pone la información de los objetos ubicados en el mapa.
- Windows: Acá esta todo lo que es visual y su interacción con el usuario.

3.2.2. Clases

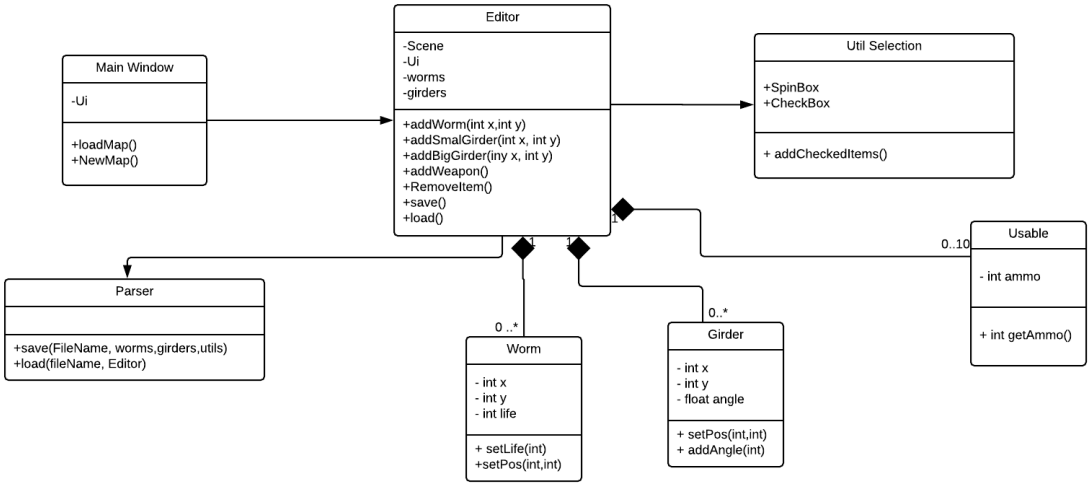

Figura 12: Diagrama de clases general del Editor de mapas

Este es un diagrama en el que se muestra la interacción de las distintas clases:

- MainWindow conoce al Editor.
- Editor conoce a UtilSelection y Parser. Además esta compuesto por Worm, Girder y Usable.

Nombre	**MainWindow**
Responsabilidad	Es la pantall de inicio del programa. En el se peude crear un nuevo mapa o editar uno existente
Atributos	- Ui
Metodos	+ loadMap(): Se carga un mapa ya existente para la continuación de su edición + newMap(): Crea un nuevo mapa a editar

Nombre	**Parser**
Responsabilidad	La responsabilidad de esta clase es la de guardar la configuracion de un mapa creado, asi como la de cargar uno para continuar su edicion
Atributos	
Metodos	+ load(): Caraga un mapa que tenga la configuracion YAML + save: Guarda un mapa con Configuracion YAML

Nombre	**Editor**
Responsabilidad	Su responsabilidad es la de ir mostrando el mapa a medida que se edita para dar una idea de como quedaria al momento de guardar. Tambien impide la creacion de un mapa si no se cumplen ciertas condiciones, como que haya menos gusanos que la cantidad maxima de jugadores
Atributos	-Scene -Ui -worms -girder -usables
Metodos	+addWorm(int x, int y): Agrega un gusano en la posicion X Y +addSmallGirder(int x, int y): Agrega una viga de 3 metros en la posicion X Y +addBigGirder(int x, int y): Agrega una viga de 6 metros en la posicion X Y +addWeapon(int id, int ammo): Agrega un arma +removeItem(): Remueve un item selecionado del mapa +save(): Guarda la configuracion del mapa +load(): Carga un mapa

Nombre	**UtilSelection**
Responsabilidad	Su responsabilidad esta en mostrar el total de armas disponibles para que se decida cuales van a ser usadas en el juego, al igual que su cantidad.
Atributos	+SpinBox +CheckBox -Ui
Metodos	addCheckedItems(): Todas las armas que están seleccionados son agregadas, y las que no removidas

Nombre	**Worm**
Responsabilidad	Su responsabilidad esta en almacenar los datos del gusano, como la vida que tiene y su posición.
Atributos	-int x -int y -int life
Metodos	+ setLife(int life): Cambia la vida del Worm +addPos(int x,int y): Suma los valores a la posicion del Worm

Nombre	**Girder**
Responsabilidad	Su responsabilidad esta en almacenar los datos de la viga, como la posicion, y su angulo de inclinacion.
Atributos	-int x -int y -float angle
Metodos	+ addPos(int x, int y): Suma los valores a la posicion de la Girder + addAngle(int angle): Aumenta el angulo de la Girder

Nombre	**Usable**
Responsabilidad	Su responsabilidad esta en almacenar el dato de la munición del arma, para que sea guardado en un archivo YAML.
Atributos	-int ammo
Metodos	+getAmmo(): Retorna la cantidad de municiones del Usable

3.2.3. Diagramas de secuencia

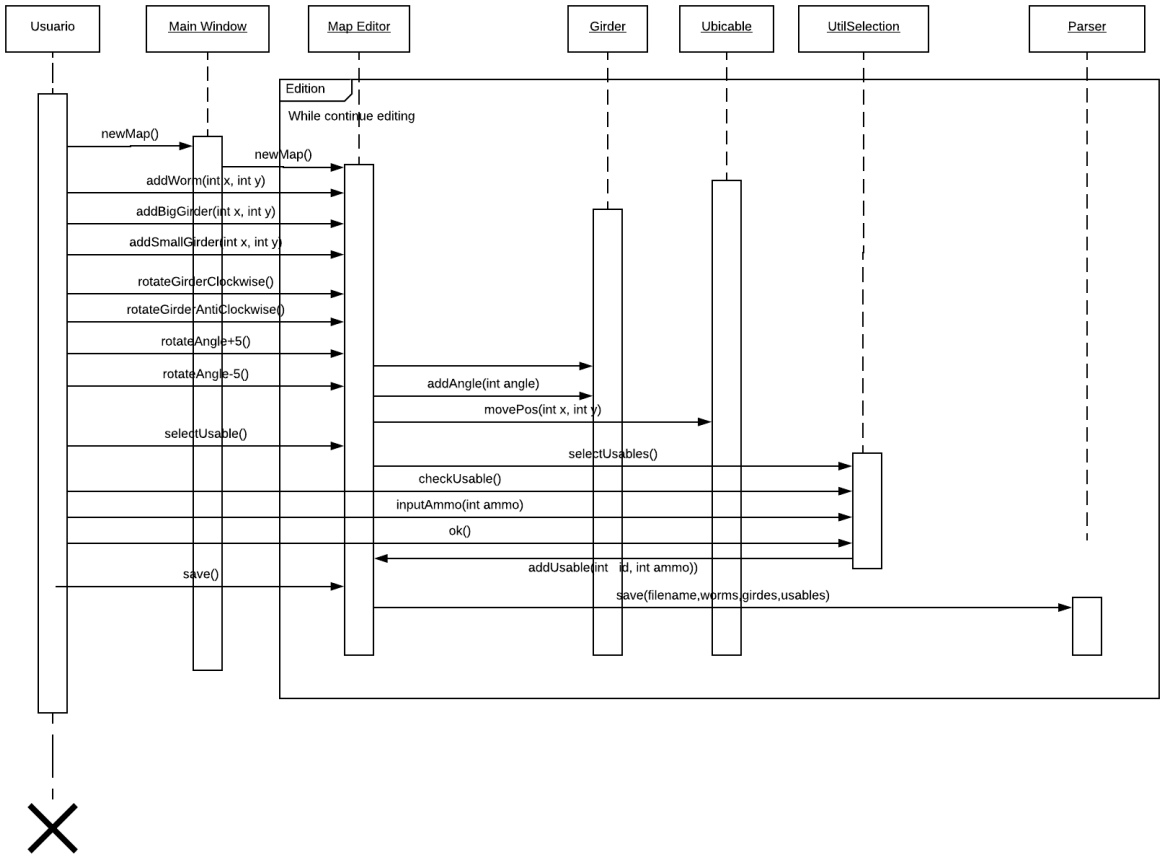

Figura 13: Secuencia de la edición de un mapa

Este es un gráfico en el que se muestra como es la creación de un nuevo mapa. Así como la interacción del Usuario con las entidades del programa.

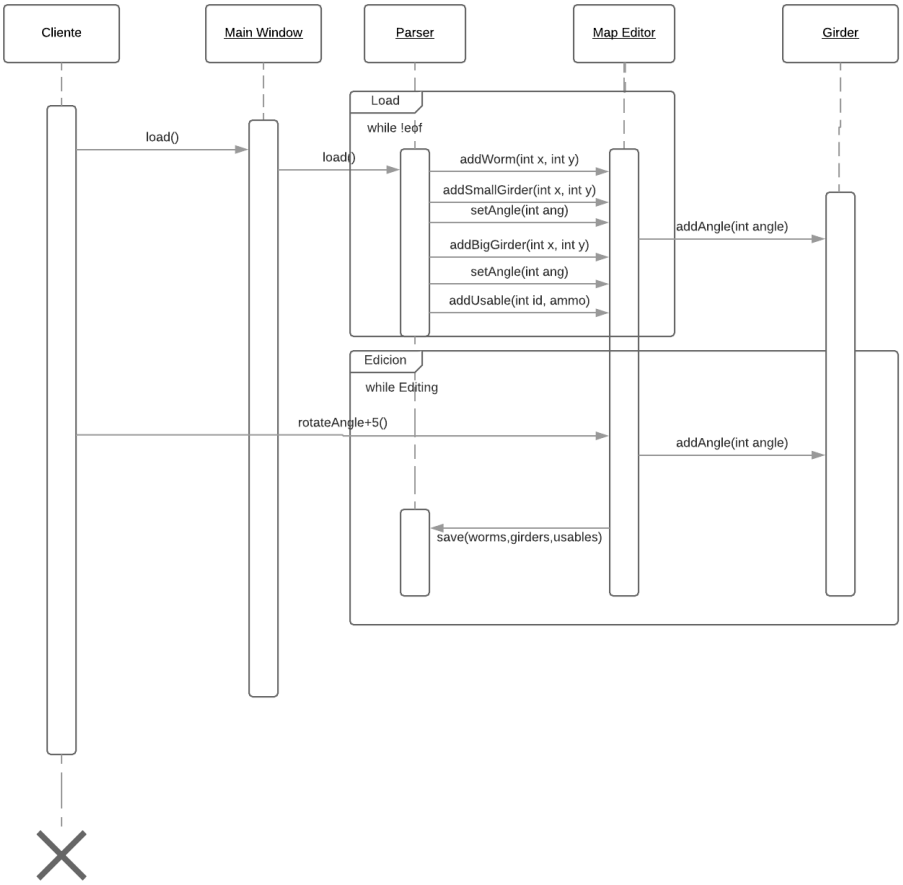

Figura 14: Secuencia de la carga de un mapa ya existente para continuar su edición

En este gráfico se ve como un usuario carga un mapa ya existente para continuar su edición. Solo se pone una operación de edición, porque estas ya fueron explicadas en el gráfico anterior.

3.3. Cliente

3.3.1. Descripción general

El Cliente es el encargado de la visualización de lo que sucede en el juego, como conectarse con el servidor y crear o unirse salas.

3.3.2. Organización

El cliente consiste en los siguientes paquetes:

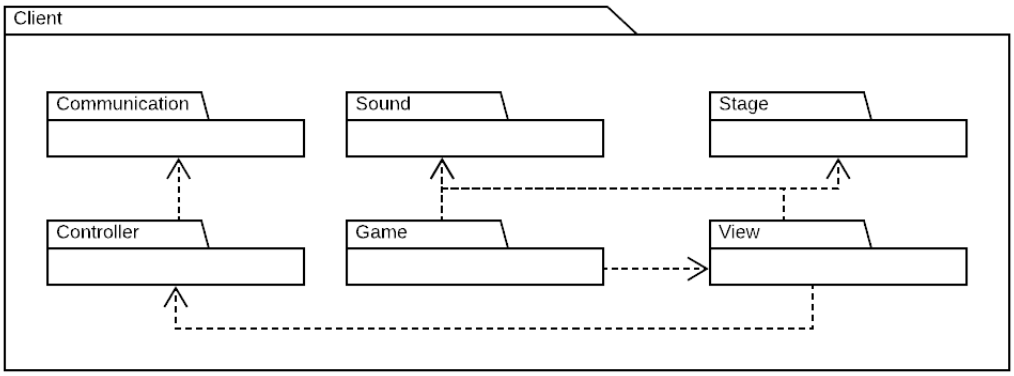

Figura 15: Paquetes del cliente y sus relaciones

Donde cada paquete se detallará a continuación:

- **Communication**: Contiene el protocolo con el que se comunica el cliente con servidor.
- **Controller**: Se encarga de redireccionar(mediante señales) correctamente los mensajes recibidos del servidor al hilo main y su clase correspondiente.
- **Sound**: Contiene las clases de Sonido a ser usadas en el Juego.
- **Game**: Contiene todos los objetos que estan presentes en el Stage y contiene la lógica para interpretar que hacer mediante señales emitidas por el Controller.
- **Stage**: Contiene todo el escenario y la camara que permite visualizarlo.
- **View**: Contiene todas las ventanas utilizadas en el juego.

3.3.3. Clases

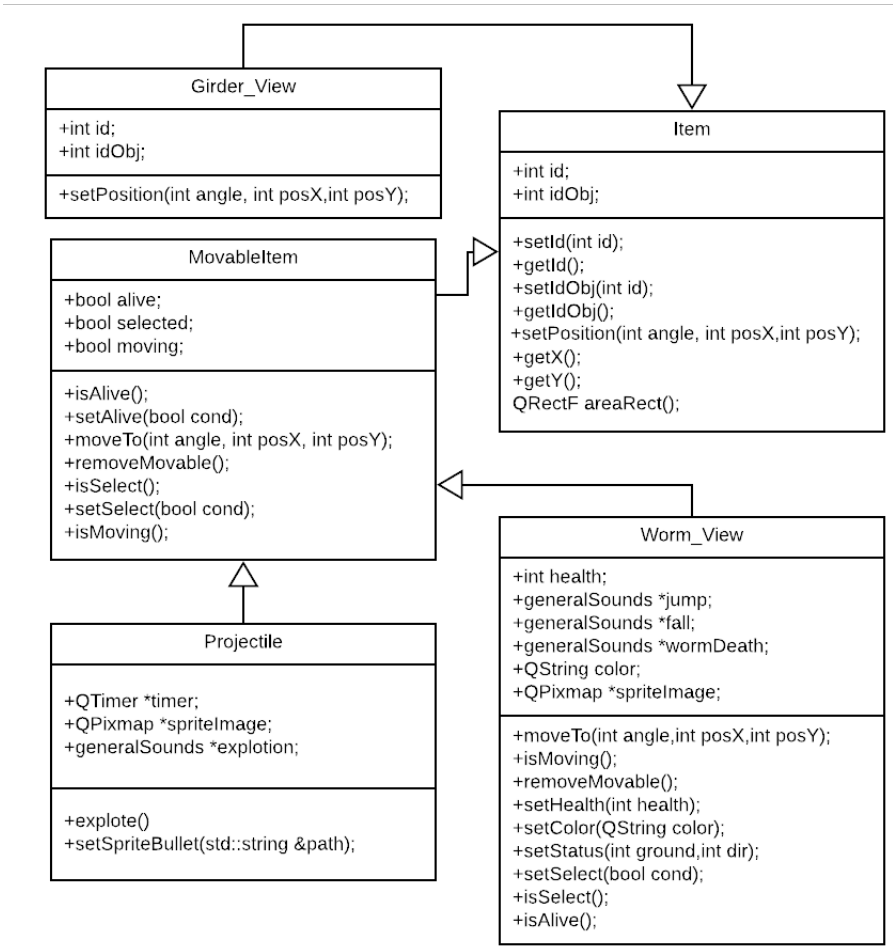

Figura 16: Diagrama de clases de Item y sus relaciones.

En dicho diagrama se puede ver la relación entre las clases mostradas. Worm y Projectile son Item del tipo MovableItem el cual permite saber si esta vivo, si se esta moviendo y si esta o no seleccionado. Por otro lado Girder es simplemente un Item.

Nombre	**Girder_View**
Responsabilidad	Representa visualmente a la Viga.
Atributos	+int id; +int idObj;
Métodos	+setPosition(int angle,int posX, int posY): Posiciona al objeto en posX posY con el angulo pasado.

Cuadro 23: Clase Girder View

Nombre	**MovableItem**
Responsabilidad	Representa a un item movible.
Atributos	+bool alive; +bool select; +bool moving;
Métodos	+isAlive() : Devuelve un booleando que representa si el item esta vivo. +setAlive(bool cond) : Seteo la condicion de vivo. +moveTo(int angle, int posx, int posy) : Muevo el item a la posicion y angulo indicado. +removeMovable() : Remueve visualmente al item. +isSelect() : Devuelve un booleano en función si está seleccionado el item. +setSelect(bool cond) : Seteo la selección del item. +isMoving() : Devuelve un booleano si se esta moviendo.

Cuadro 24: Clase MovableItem

Nombre	**Item**
Responsabilidad	Representa a un item.
Atributos	+int id; +int idObj;
Métodos	+setId(int id) : Seteo el id del item. +getId() : Devuelve el id del item. +setIdObj(int id) : Seteo el idObj del Item. +getIdObj() _ Devuelve el idObj del item. +setPosition(int angle, int x, int y) : Posiciono al objecto con angulo segun los parametros pasados. +getX() : Devuelve la posicion X del item. +getY() : Devuelve la posicion Y del item. +areaRect() : Devuelve el area que ocupa el item.

Cuadro 25: Clase Item

Nombre	**Projectil**
Responsabilidad	Clase genérica que representa a los projectiles.
Atributos	+QTimer *timer; +QPixmap *spriteImage; +gemeralSounds *explotion;
Métodos	+explote() : Realiza la explosion visualmente. +setSpriteBullet(std::string &path) : Setea el sprite a usar como imagen de projectil.

Cuadro 26: Clase Projectile

Nombre	**Worm_View**
Responsabilidad	Representa la visualizacion del worm.
Atributos	+int health; +QPixmap *spriteImage; +QString color; +generalSounds *fall; +generalSounds *jump; +generalSounds *wormDeath;
Métodos	+setHealth(int health) : Setea la vida del worm. +setColor(QString color) : Setea el color del indicador de vida. +setStatus(int ground, int dir) : Setea el estado del worm, si esta o no en tierra y su dirección.

Cuadro 27: Clase Worm View

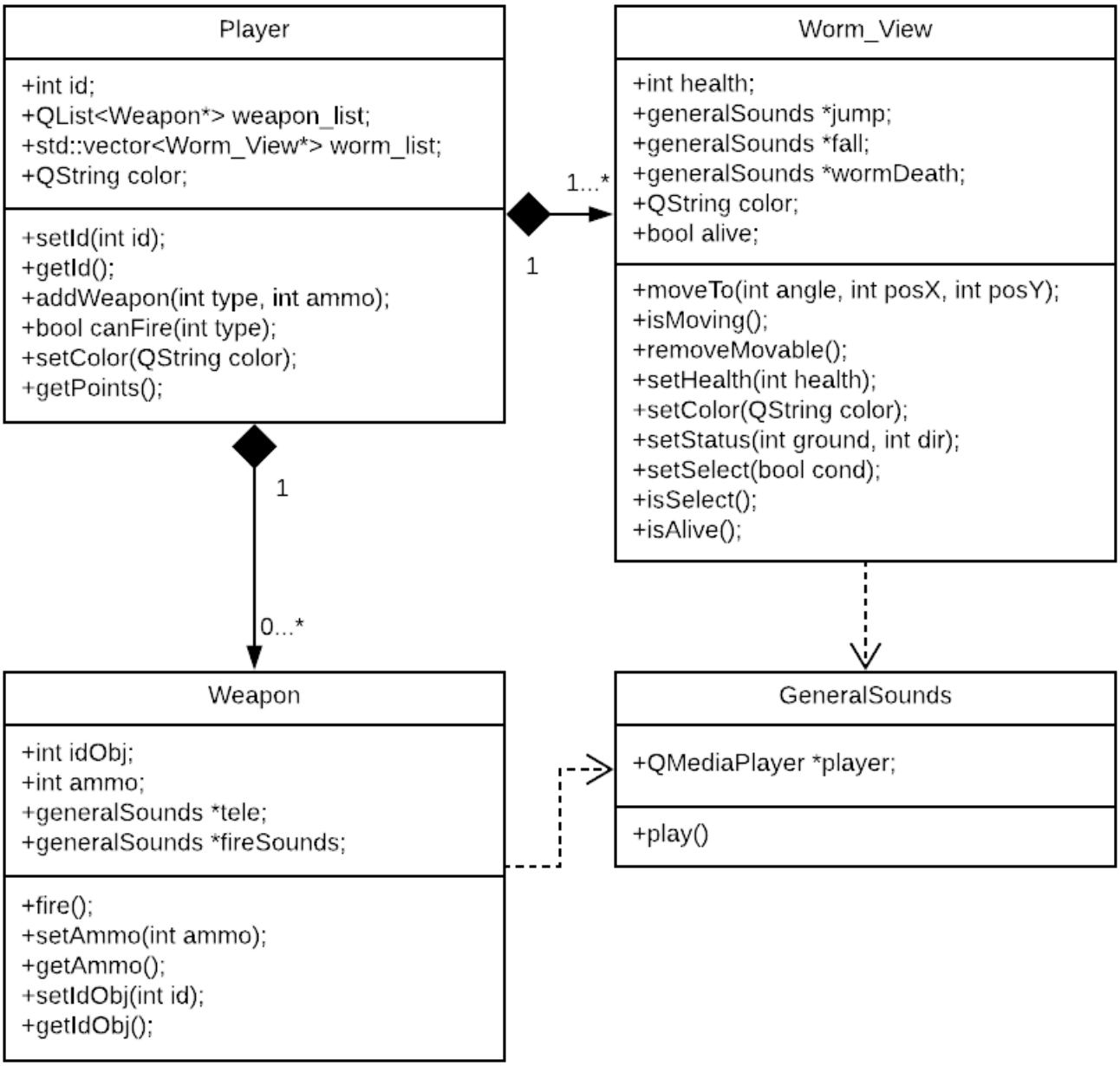

Figura 17: Diagrama de clases de Player y sus relaciones.

En dicho diagrama se puede ver en particular la clase Player y sus relaciones. Player puede contener o no armas, pero si o si contiene al menos un Worm. A su vez en éste mismo se muestra la relación del sonido, donde Worm tiene sonido al saltar y morir y las armas tienen sonido al ser lanzadas.

Nombre	**Player**
Responsabilidad	Representación del jugador.
Atributos	+int id; +QList<Weapon*>weapon_list; +std::vector<Worm_View*>worm_list; +QString color;
Métodos	+setId(int id) : Setea el id del player. +getId() : Devuelve el id del player. +addWeapon(int type, int ammo) : Agrega el arma para ser usada por el jugador y sus worms. +canFire(int type) : Devuelve si el arma puede realizar un disparo. +setColor(QString color) : Setea el color que lo representa. +getPoints() : Devuelve la cantidad de puntos que tiene el jugador.

Cuadro 28: Clase Player

Nombre	**Weapon**
Responsabilidad	Representacion genérica de las armas.
Atributos	+int idObj; +int ammo; +generalSounds *tele; +generalSounds *fireSounds;
Métodos	+setIdObj(int id) : Setea el id del arma. +getIdObj() : Devuelve el id del arma. +setAmmo(int ammo) : Setea la cantidad de munición que tiene el arma. +getAmmo() : Devuelve la cantidad de munición disponible.

Cuadro 29: Clase Weapon

Nombre	**GeneralSounds**
Responsabilidad	Representa los sonidos generales.
Atributos	+QMediaPlayer *player;
Métodos	+play() : Ejecuta el sonido

Cuadro 30: Clase GeneralSounds

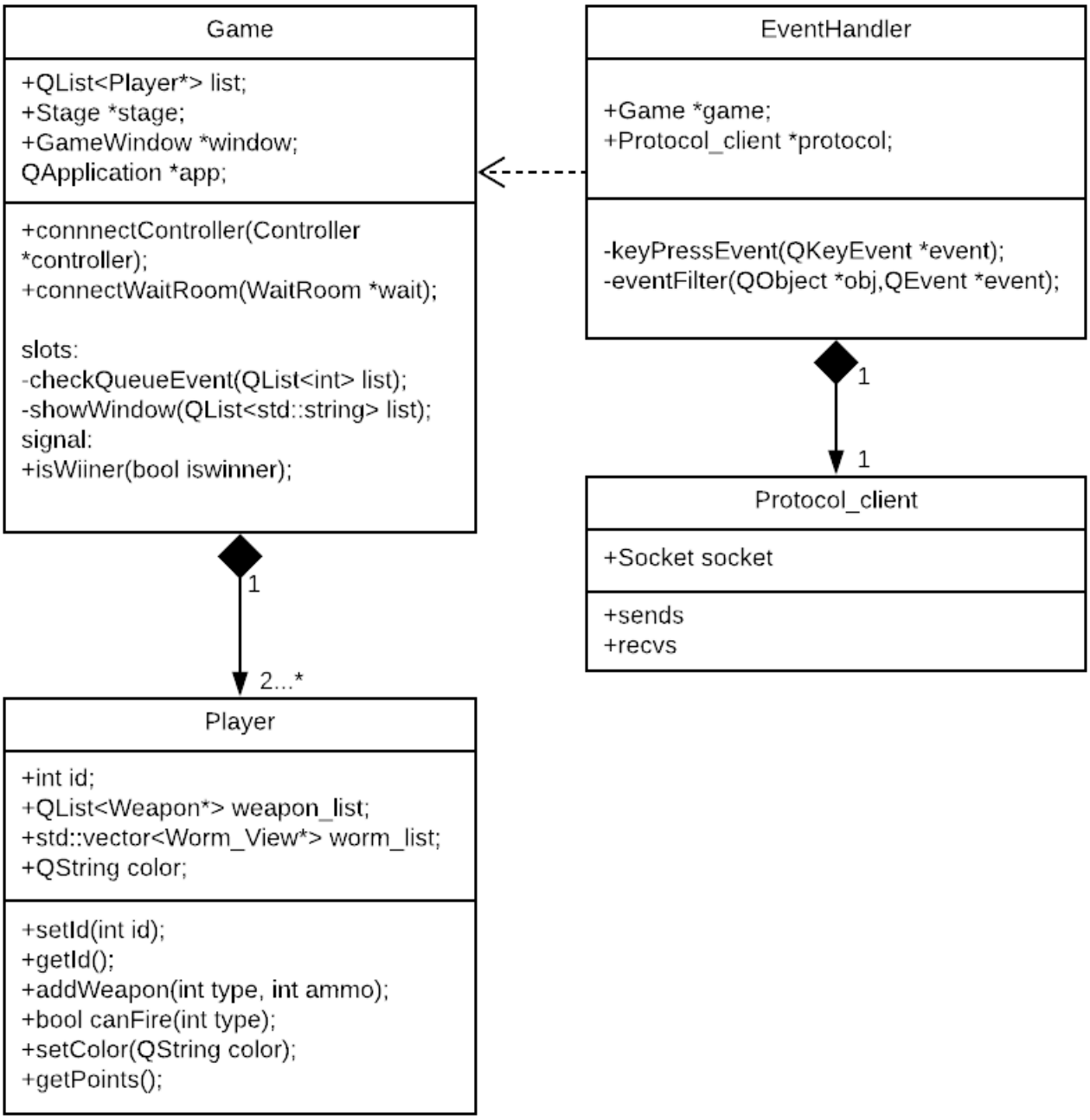

Figura 18: Diagrama de clases de Game y sus relaciones.

En el diagrama anterior se puede ver la clase Game y alguna de sus relaciones. EventHandler es una clase que conoce y utiliza al Game, ya que la misma se encarga de los eventos de tipo teclado en particular. También se puede observar que Game solo debería existir si contiene al menos 2 Players.

Nombre	**Game**
Responsabilidad	Representación del juego.
Atributos	+QList<Player*>player_list; +Stage *stage; +GameWindow *window; +QApplication *app;
Métodos	+connectController(Controller *controller) : Conecta la clase Game con la clase Controller, en la cual se transmiten señales conectadas a slots. +connectWaitRoom(WaitRoom *wait) : Conecta la clase Game con la clase WaitRoom, la cual le avisa que se inicio el juego. +checkQueueEvent(QList<int>list) : Slot que está conectado al controller que genera eventos recibidos del servidor. +showWindow(QList<std::string>list) : Slot que está conectado al WaitRoom que emite la señal de inicio.

Cuadro 31: Clase Game

Nombre	**EventHandler**
Responsabilidad	Se encarga de capturar los eventos de teclado.
Atributos	+Game *game; +Protocol_client *protocol;
Métodos	+keyPressEvent(QKeyEvent *event) : Interpreta la tecla presionada y ejecuta su debida acción. +eventFilter(QObject *obj, QEvent *event) : Filtra el evento ocurrido.

Cuadro 32: Clase EventHandler

Nombre	**Protocol_client**
Responsabilidad	Clase que representa un protocolo de comunicación de datos desde el lado del cliente. Hereda de Protocol que guarda el socket
Atributos	
Métodos	+ recvWormId(id, wormid, health): Recibe el worm que le pertenecerá al jugador con dicho id. + recvUsableId(id, ammo): Recibe las armas que el jugador puede utilizar junto a la cantidad de municiones. + recvPlayerId(id): Recibe el id del jugador. ...

Cuadro 33: Clase Protocol client

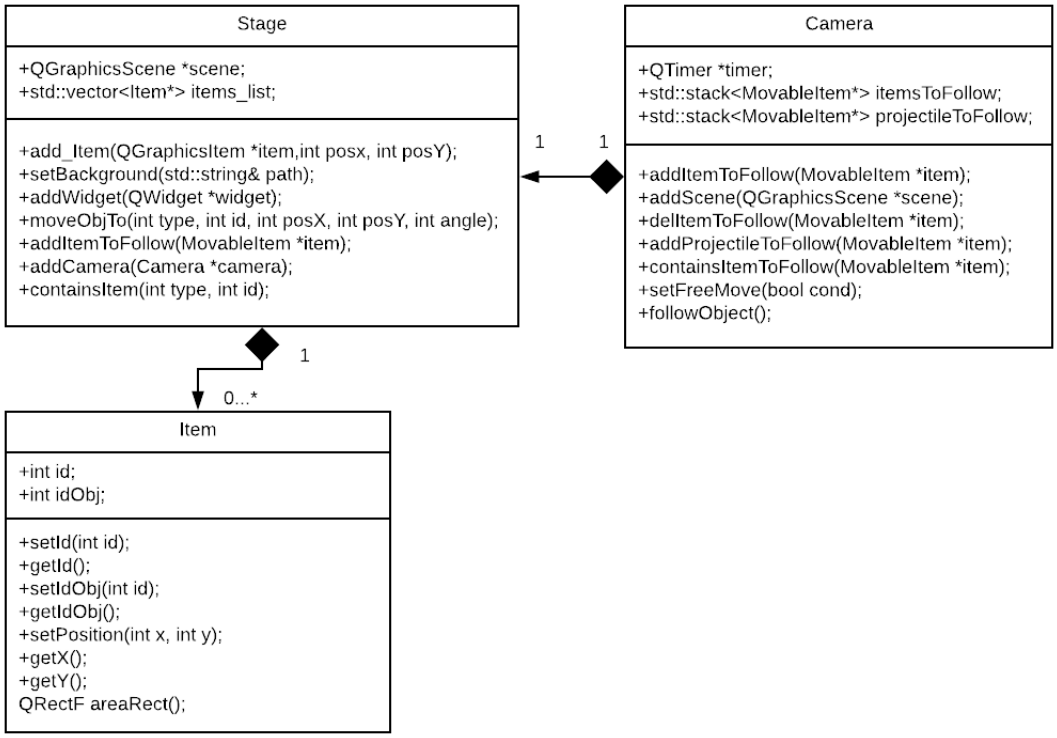

Figura 19: Diagrama de clases del Stage y sus relaciones.

En el diagrama anterior se muestra como es el Stage y por qué está compuesto. Como describe la imagen un Stage puede contener o no Item, pero si o si debe contener una sola Camera.

Nombre	**Stage**
Responsabilidad	Clase que representa el escenario, en la cual todos los item son colocados. Para la visualizacion es necesario una Camara.
Atributos	+QGraphicsScene *scene; +std::vector<Item*>item_list;
Métodos	+add_Item(QGraphicsItem *item,int posx, int posY) : Agrega un item en la posición dada. +setBackground(std::string& path) : Setea la imagen de Background. +addWidget(QWidget *widget) : Agrega un widget. +moveObjTo(int type, int id, int posX, int posY, int angle) : Mueve al item identificado por tipo e id, en caso de no conenerlo, no hace nada. +addItemToFollow(MovableItem *item) : Se agrega un item movible a ser seguido por la camara. +addCamera(Camera *camera) : Se guarda una referencia de la Camara. +containsItem(int type, int id) : Devuelve un booleano si contiene al item representado por el tipo e id.

Cuadro 34: Clase Stage

Nombre	**Camera**
Responsabilidad	Clase que representa el escenario, en la cual todos los item son colocados. Para la visualizacion es necesario una Camara.
Atributos	+QTimer *timer; +std::stack<MovableItem*>itemsToFollow; +std::stack<MovableItem*>projectileToFollow;
Métodos	+addItemToFollow(MovableItem *item) : Agrega item a seguir. +addScene(QGraphicsScene *scene) : Se vincula la camara con el escenario. +delItemToFollow(MovableItem *item) : Se quita el item a seguir. +addProjectileToFollow(MovableItem *item) : Se Agrega un projectil a seguir. +containsItemToFollow(MovableItem *item) : Devuelve un booleano si contiene al item pasado por parámetro. +setFreeMove(bool cond) : Setea la camara libre movimiento. +followObject() : Sigue un objeto.

Cuadro 35: Clase Camera

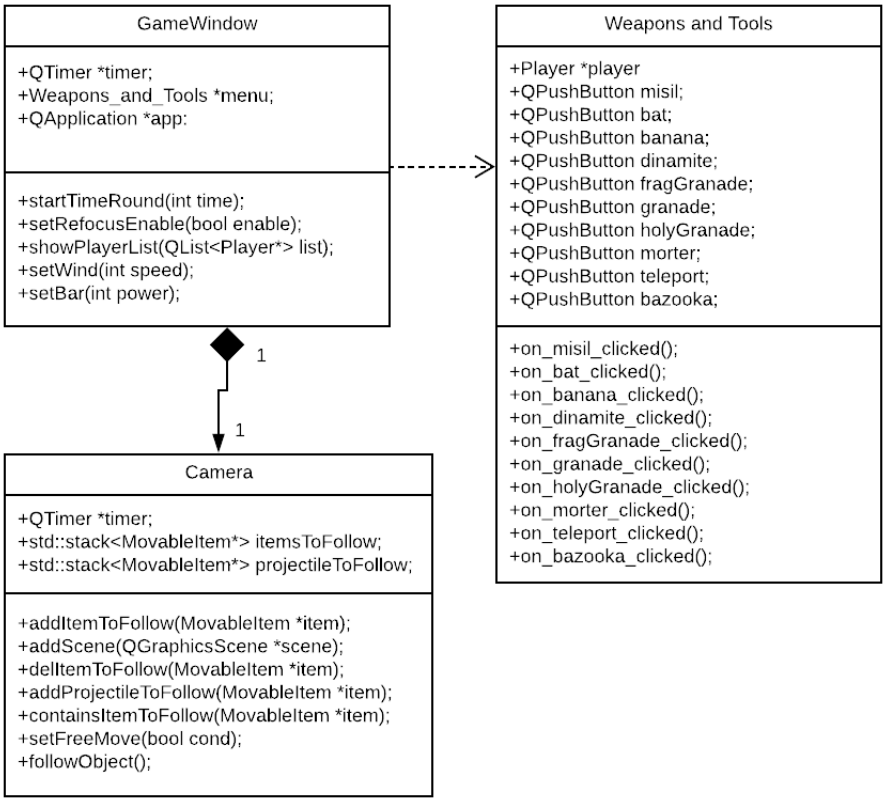

Figura 20: Diagrama de clase de GameWindow y sus relaciones

En el diagrama anterior se puede observar como esta compuesta la ventana en la que se visualiza todo el juego, la que contiene el menu para seleccionar armas, donde se puede ver a los Worms e información de los jugadores.

Nombre	**Weapons and Tools**
Responsabilidad	Ventana que contiene las armas y herramientas disponibles
Atributos	+Player *player +QPushButton misil; +QPushButton bat; +QPushButton banana; +QPushButton dinamite; +QPushButton fragGranade; +QPushButton granade; +QPushButton holyGranade; +QPushButton morter; +QPushButton teleport; +QPushButton bazooka;
Métodos	+on_misil_clicked() : Evento click en boton de AirAttack. +on_bat_clicked() : Evento click en boton de Baseball Bat. +on_banana_clicked() : Evento click en boton de Banana. +on_dinamite_clicked() : Evento click en boton Dinamite. +on_fragGranade_clicked() : Evento click en boton Frag Granade. +on_granade_clicked() : Evento click en boton Granade. +on_holyGranade_clicked() : Evento click en boton Holy Granade. +on_morter_clicked() : Evento click en boton Morter. +on_teleport_clicked() : Evento click en boton Teleport. +on_bazooka_clicked() : Evento click en boton bazooka. Todos los eventos, setean de ser posible el arma en el worm y cargan en GameWindow la informacion para disparar dicha arma.

Cuadro 36: Clase Weapons and Tools

Nombre	**GameWindow**
Responsabilidad	Ventana en la cual se desenvuelve el juego.
Atributos	+QTimer *timer; +Weapons_and_Tools *menu; +QApplication *app:
Métodos	+startTimeRound(int time) : Inicia el tiempo de partida. +setRefocusEnable(bool enable) : Setea la Camara en seguir algun objeto en movimiento. +showPlayerList(QList<Player*>list) : Muestra en un panel la lista de jugadores y sus puntajes. +setWind(int speed) : Setea la velocidad del viento a mostrar. +setBar(int power) : Setea la potencia con la que se esta por o se disparó.

Cuadro 37: Clase GameWindow

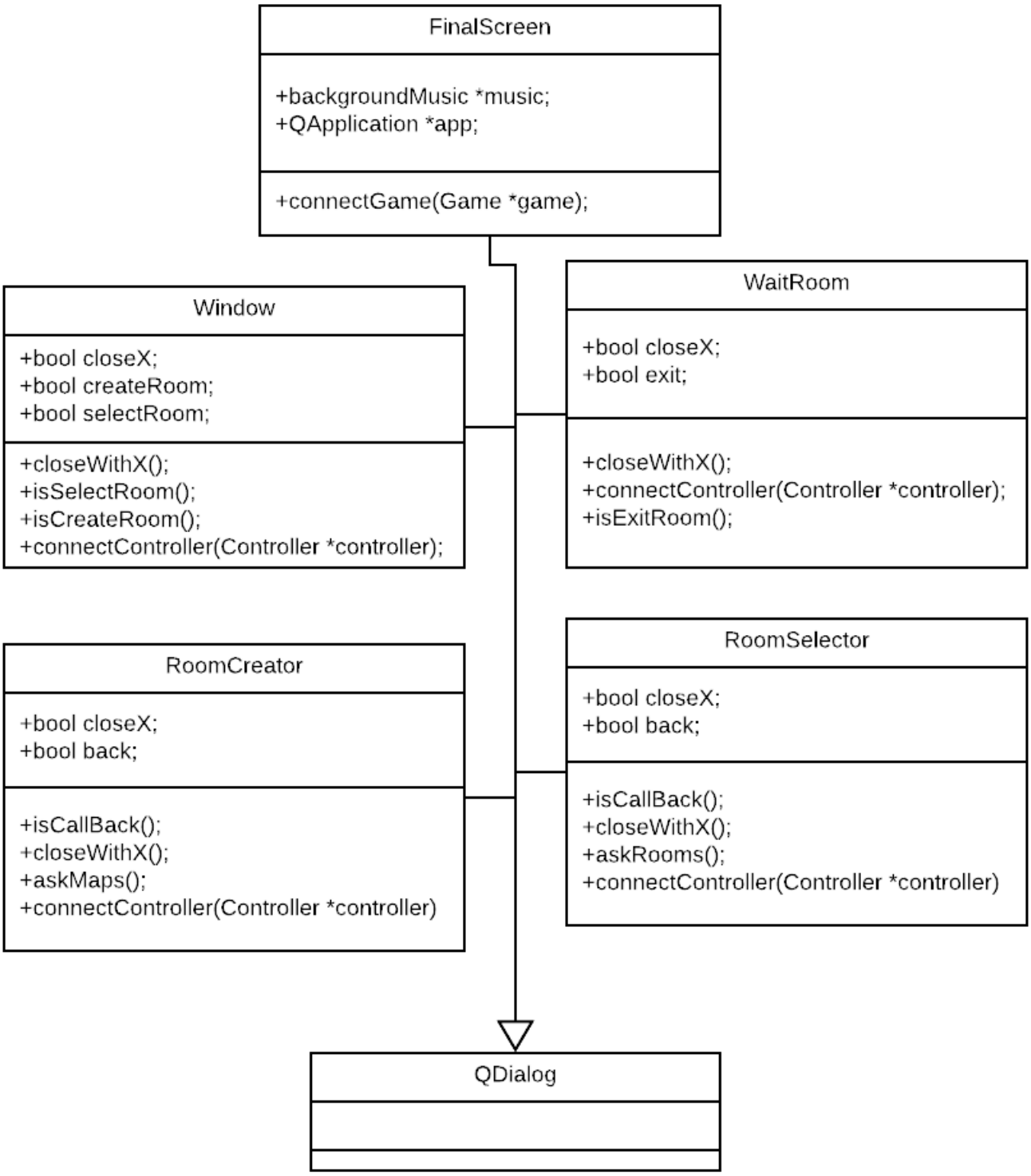

Figura 21: Diagrama de las Ventanas.

En la figura anterior se muestran las ventanas que permiten la conexión con el servidor, creación de salas, conexión a salas existentes, sala de espera, y pantalla final en la que se muestra si fuiste ganador o perdedor. Para entender mejor la relación entre las mismas se recomienda ver los diagramas de secuencia a continuación.

Nombre	**Window**
Responsabilidad	Ventana en la cual se selecciona si uno desea crear o unirse a una sala.
Atributos	+bool closeX; +bool createRoom; +bool selectRoom;
Métodos	+closeWithX() : Indica si la ventana fue cerrada con el boton x del vertice superior izquierdo. +isSelectRoom() : Indica si el boton presionado fue de selección de sala. +isCreateRoom() : Indica si el boton presionado fue de creación de sala. +connectController(Controller *controller) : Conecta señales y slots entre el controller y la ventana.

Cuadro 38: Clase Window

Nombre	**RoomCreator**
Responsabilidad	Ventana en la que se mostrarán los mapas disponibles sobre los cuales se puede crear una sala.
Atributos	+bool closeX; +bool back;
Métodos	+isCallBack() : Indica si fue presionado el boton de ir hacia atras. +closeWithX() : Indica si fue cerrada la ventana con el boton x del vértice superior izquierdo. +askMaps() : Se comunica con el Server para pedir los mapas disponibles a mostrar. +connectController(Controller *controller) : Conecta slots y señales de la ventana con el controller.

Cuadro 39: Clase RoomCreator

Nombre	**RoomSelector**
Responsabilidad	Ventana en la que se mostrarán las salas disponibles a las que se podrá unir.
Atributos	+bool closeX; +bool back;
Métodos	+isCallBack() : Indica si fue presionado el boton de ir hacia atras. +closeWithX() : Indica si fue cerrada la ventana con el boton x del vértice superior izquierdo. +askRooms() : Se comunica con el Server para pedir las salas disponibles a mostrar. +connectController(Controller *controller) : Conecta slots y señales de la ventana con el controller.

Cuadro 40: Clase RoomCreator

Nombre	**WaitRoom**
Responsabilidad	Ventana para esperar a que la sala se llene con la cantidad de jugadores necesaria para iniciar la partida.
Atributos	+bool closeX; +bool exit;
Métodos	+closeWithX() : Indica si la ventana fue cerrada con el boton x del vértice superior izquierdo. +connectController(Controller *controller) : Conecta señales y slots entre el controller y la ventana. +isExitRoom() : Indica si se presionó el boton para salir de la sala.

Cuadro 41: Clase WaitRoom

Nombre	**FinalScreen**
Responsabilidad	Ventana final que muestra visualmente si fuiste ganador o perdedor.
Atributos	+backgroundMusic *music; +QApplication *app;
Métodos	+connectGame(Game *game) : Conecta señales y slots entre la ventana y el Game.

Cuadro 42: Clase FinalScreen

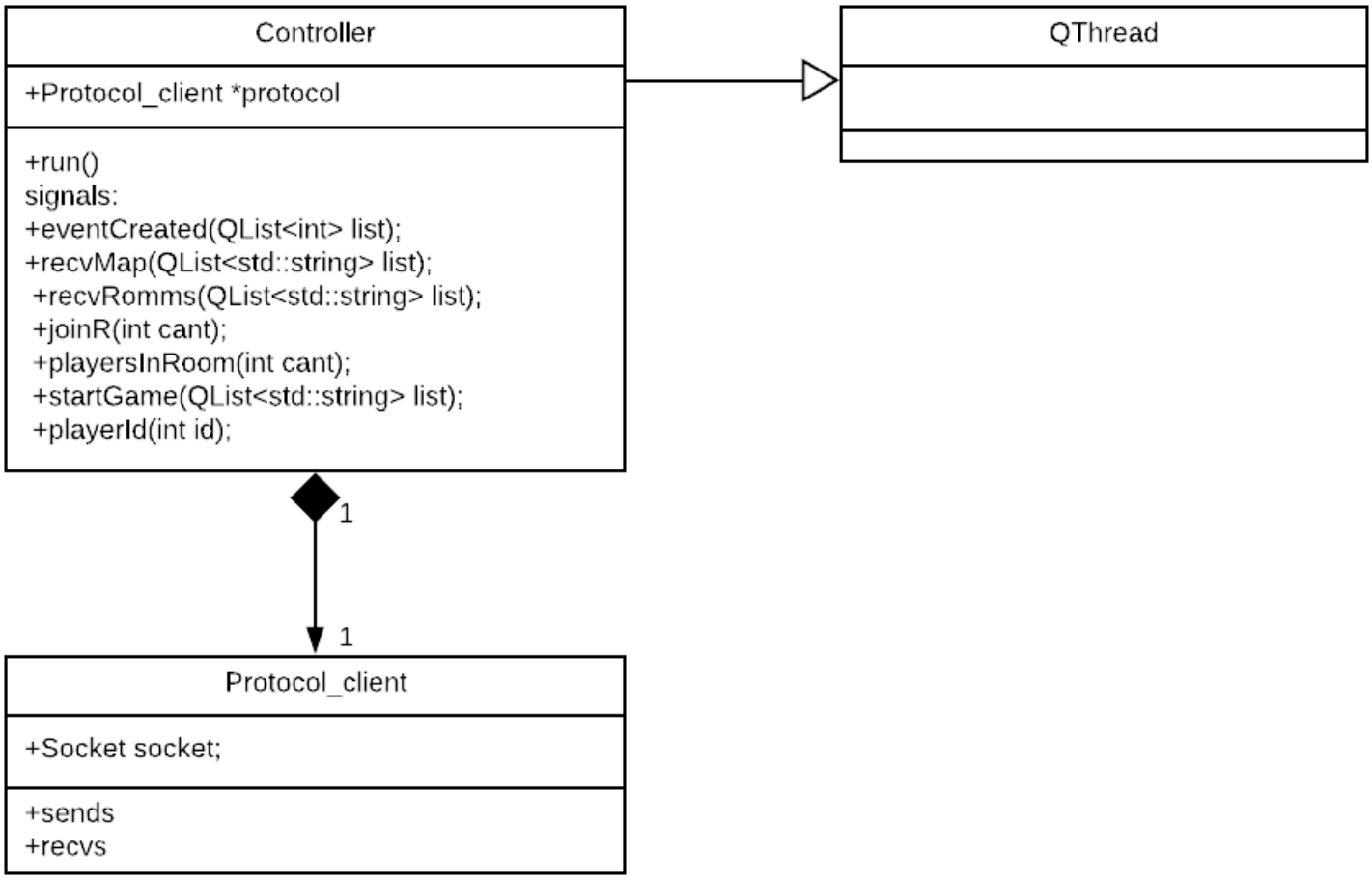

Figura 22: Diagrama del Controller y sus relaciones.

En dicho diagrama se intenta mostrar la clase Controller y sus relaciones. La clase Controller es la encargada de estar permanente a la escucha de los mensajes del servidor y emitir señales al hilo main para su procesamiento.

Nombre	**Controller**
Responsabilidad	Esta clase es la encargada de estar a la escucha de los mensajes enviados por el servidor y generar las señales correspondientes para ser recibidas por otras clases en el hilo main.
Atributos	+Protocol_client *protocol
Métodos	+run() : Método sobreescrito de la clase QThread, signals: +eventCreated(QList<int>list) : Esta señal se usa para comunicarse con la clase Game. +recvMap(QList<std::string>list) : Esta señal se comunica con la clase RoomCreator, le pasa los mapas disponibles. +recvRooms(QList<std::string>list) : Esta señal se conecta con la clase RoomSelector, le pasa las salas activas y disponibles. +joinR(int cant) : Esta señal se conecta con dos clases, RoomCreator y con RoomSelector, les indica si se pudo unir o crear la sala. +playersInRoom(int cant) : Señal que indica la cantidad de jugadores en la sala de espera. +startGame(QList<std::string>list) : Señal que indica que el juego comenzó. +playerId(int id) : Señal conectada con la ventana Window, la cual almacena el id al conectarse con el servidor.

Cuadro 43: Clase Controller

3.3.4. Diagramas de secuencia

A continuación se detallan las ventanas para la conexión al servidor

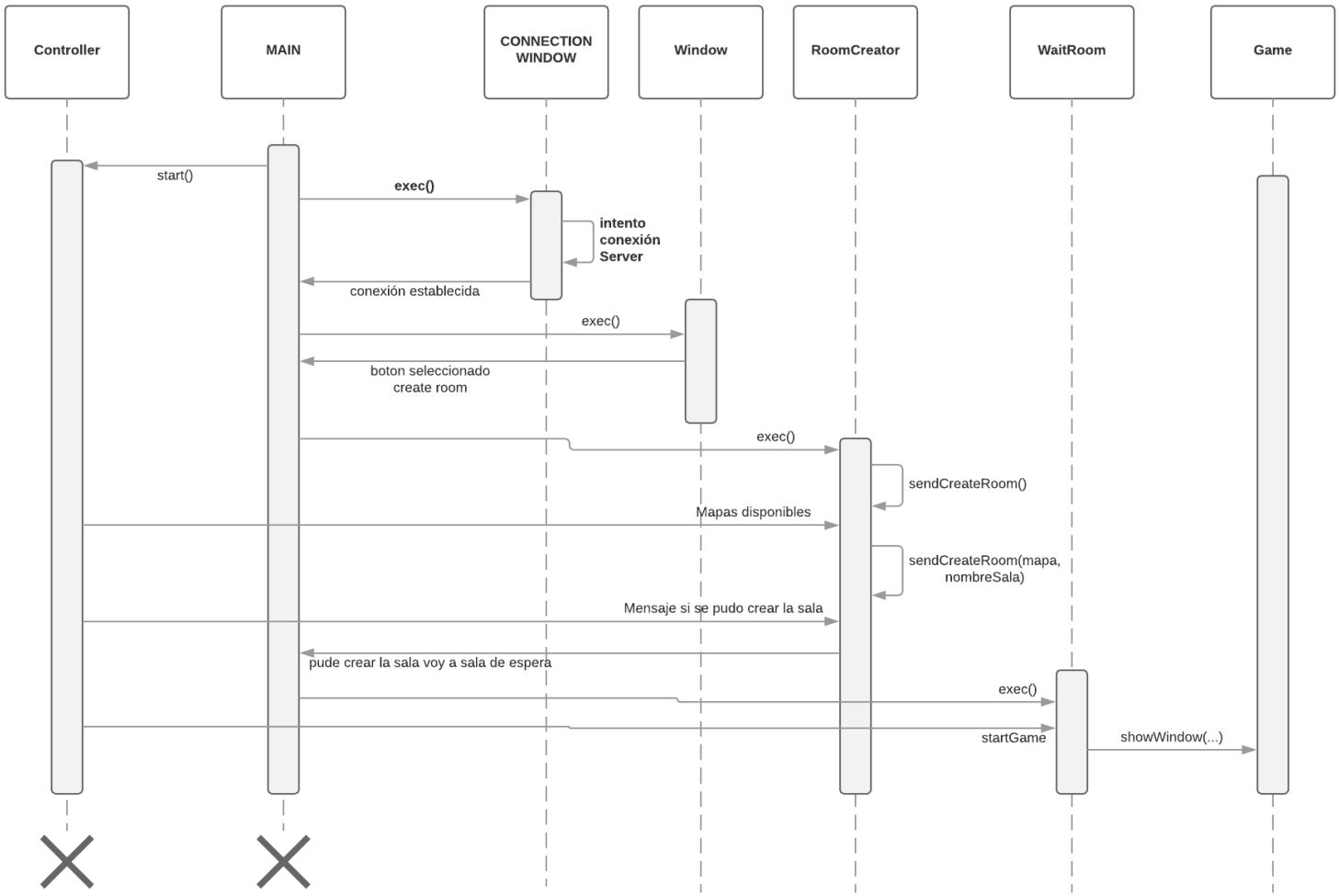

Figura 23: Diagrama secuencia - Creación de sala.

En el diagrama anterior se quiere mostrar una aproximación a como suceden los mensajes/señales entre las clases y como se va cambiando de ventana según la opción elegida, en este caso crear una sala.

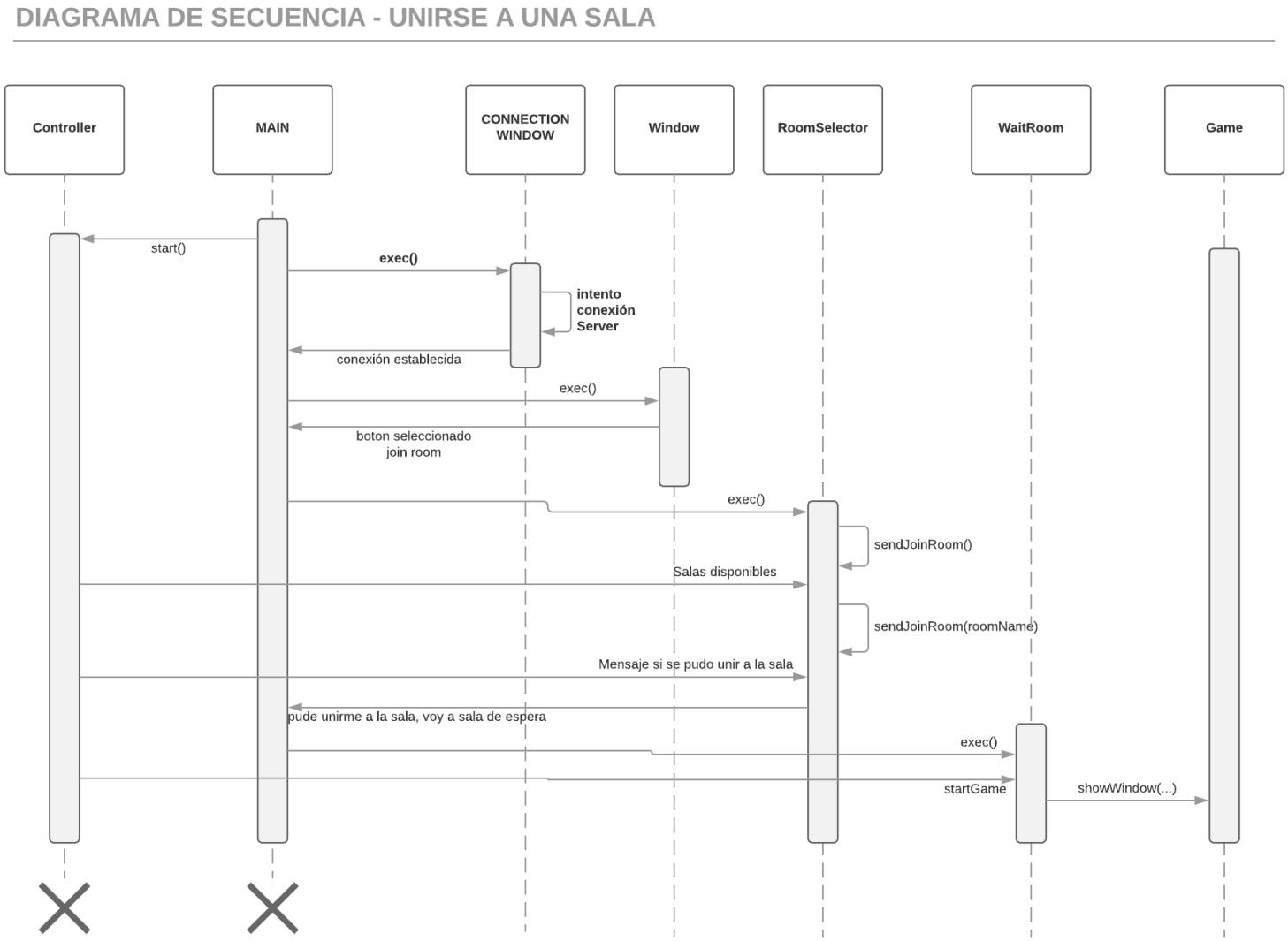

Figura 24: Diagrama secuencia - Unirse a una sala.

En el diagrama anterior se quiere mostrar una aproximación a como suceden los mensajes/señales entre las clases y como se va cambiando de ventana según la opción elegida, en este caso unirse a una sala ya existente.

4. Common

4.1. Descripción general

El paquete common contiene clases que utilizan tanto el cliente como el servidor, a continuación, se dará un breve detalle de cada una de ellas:

4.2. Clases

Nombre	**Counter**
Responsabilidad	Representa un contador que se le puede modificar la duración.
Atributos	- counter: int - m: mutex
Métodos	+ set_time(secs): Setea la nueva duración del contador. + start_counting(): Arranca a contar. + stop(): Detiene el contador.

Cuadro 44: Clase contador

Nombre	**BlockQueue**
Responsabilidad	Una cola bloqueante que espera a que haya algún elemento en la cola.
Atributos	- queue: queue - m: mutex - cond_var: condition_variable
Métodos	+ push(T&&): Agrega un objeto a la cola. + pop: Devuelve el objeto que se encuentre primero en la cola.

Cuadro 45: Clase BlockQueue

Nombre	**Protocol**
Responsabilidad	Representa un protocolo de comunicación.
Atributos	- conexion: Socket&
Métodos	+ send_string(str): Envía un string. + send_char(char): Envía un char. + send_cmd(cmd&): Envía un comando. + send_type_obj(type&): Envía un tipo de objeto del juego. + send_usable(usable&): Envía un usable para el jugador. + send_int_signed(int): Envía un entero con signo. + send_int_unsigned(uint): Envía un entero sin signo. + recv_string(): Recibe un string. + recv_char(): Recibe un char. + recv_cmd(): Recibe un comando. + recv_type_obj: Recibe un tipo de objeto del juego. + recv_usable(usable&): Recibe un usable para el jugador. + recv_int_signed(int): Recibe un entero con signo. + recv_int_unsigned(uint): Recibe un entero sin signo.

Cuadro 46: Clase Protocol

Nombre	**Thread**
Responsabilidad	Clase para manejar hilos por herencia.
Atributos	- thread: thread
Métodos	+ start(): Inicia el hilo aparte. + join(): Espera a que el hilo termine y lo finaliza. + run(): Acción que va a ejecutar el hilo.

Cuadro 47: Clase Thread

Nombre	**Socket**
Responsabilidad	Clase que representa un socket.
Atributos	- skt: int
Métodos	+ aceptar(): Devuelve un socket de un cliente que se conecte. + desconectar(): Desconecta el socket. + recibir(buff, size): Recibe un mensaje de un determinado tamaño. + enviar(buff, size): Envía un mensaje de un determinado tamaño.

Cuadro 48: Clase Socket

5. Protocolo de comunicación

A continuación, se mostrarán todos los mensajes (junto a los paquetes) que se comunican entre cliente y servidor para el funcionamiento del juego vía TCP/IP:

Servidor —>Cliente	**Cliente —>Servidor**
recv_cmd(): Recibe un comando del jugador.	recv_cmd(): Recibe un comando del servidor.
sendPosition(): Envía la posición y rotación de un objeto del juego.	recvPosition(): Recibe la posición y rotación de un objeto del juego.
sendWormId(): Envía un worm que le corresponde al jugador junto a la vida del mismo.	recvWormId(): Recibe un worm que le corresponde al jugador junto a la vida del mismo.
sendUsableId(): Envía un arma que le corresponde al jugador junto a la munición de la misma.	recvUsableId(): Recibe un arma que le corresponde al jugador junto a la munición de la misma.
sendPlayerId(): Envía el id de jugador que lo representa	recvPlayerId(): Recibe el id de jugador que lo representa.
sendRemove(): Envía un objeto que ha sido removido del juego.	recvRemove(): Recibe un objeto que ha sido removido del juego.
sendGameEnd(): Notifica al jugador que el juego ha terminado.	recv_cmd(): Recibe el comando de que el juego ha finalizado.
sendActualPlayer(): Notifica el id del jugador actual junto al worm que debe de utilizar.	recvActualPlayer(): Recibe el id del jugador actual junto al worm que debe de utilizar.
sendWinner(): Notifica el ganador de la partida.	recvWinner(): Recibe el ganador de la partida.
sendWormHealth(): Notifica la vida actual de un worm que ha sido dañado.	recvWormHealth(): Recibe la vida actual de un worm que ha sido dañado.
sendDisconnect(): Notifica que se va a desconectar al jugador.	recv_cmd(): Recibe el comando de que se va a desconectar al jugador.
sendCouldJoinRoom(): Notifica si el jugador se pudo unir a una sala.	recvCouldJoinRoom(): Recibe si el jugador se pudo unir a una sala.
sendPlayersInRoom(): Envía la cantidad de jugadores de la sala en la que se encuentra el jugador.	recvPlayersInRoom(): Recibe la cantidad de jugadores de la sala en la que se encuentra el jugador.
sendStartGame(): Notifica que el juego va a comenzar y envía el fondo de pantalla del mismo.	recvBackground(): Recibe el fondo de pantalla para el comienzo de una partida.
sendRooms(): Envía las salas disponibles para que el jugador se pueda unir a alguna de ellas.	recvRooms(): Recibe las salas disponibles para que el jugador se pueda unir a alguna de ellas.
sendMaps(): Envía los mapas que se pueden crear.	recvMaps(): Recibe los mapas que se pueden crear.
sendWindParams(): Envía el valor máximo y mínimo posible del viento en el juego.	recvWindParams(): Recibe el valor máximo y mínimo posible del viento en el juego.
sendWindSpeed(): Envía el valor actual del viento.	recvWindSpeed(): Recibe el valor actual del viento.
sendWormStatus(): Envía el estado en el que se encuentra el worm en ese instante, ya sea hacía donde está mirando y si se encuentra o no sobre una viga.	recvWormStatus(): Recibe el estado en el que se encuentra el worm en ese instante, ya sea hacía donde está mirando y si se encuentra o no sobre una viga.
recvRoom(): Recibe la sala a la que se quiere unir el jugador.	sendSelectRoom(): Envía la sala a la que se quiere unir el jugador.
recvMove(): Recibe un comando de movimiento para el worm actual por parte del jugador.	sendMove(): Envía un comando de movimiento para el worm actual.
recvAttack(): Recibe una posición y el arma que se desea utilizar para dispararla desde el worm actual junto a parámetros extra (Ej: Potencia de disparo).	sendAttack():Recibe una posición y el arma que se desea utilizar para dispararla desde el worm actual junto a parámetros extra (Ej: Potencia de disparo).
recvCreateRoom(): Recibe el nombre de la sala que el jugador intenta crear.	sendCreateRoom(): Envía el nombre de la sala que el jugador intenta crear.
recv_cmd(): Recibe el comando de que el jugador quiere salir de la sala.	sendExitRoom(): Envía que el jugador se quiere salir de la sala a la que pertenece.

Cuadro 49: Protocol de comunicación entre cliente y servidor