
FACULTAD DE INGENIERÍA - U.B.A.

75.42 TALLER DE PROGRAMACIÓN I

1ER. CUATRIMESTRE DE 2018

# Trabajo Práctico Final: Worms

## Documentación técnica

GRUPO: 4

CORRECTOR: PABLO DANIEL ROCA

ALEJANDRO DANERI, PADRÓN: 97839
alejandrodaneri07@gmail.com

MATIAS LEANDRO FELD, PADRÓN: 99170
feldmatias@gmail.com

AGUSTÍN ZORZANO, PADRÓN: 99224
aguszorza@gmail.com

REPOSITORIO A GITHUB:
https://github.com/AlejandroDaneri/tp-final-taller

# 1 Requerimientos de software

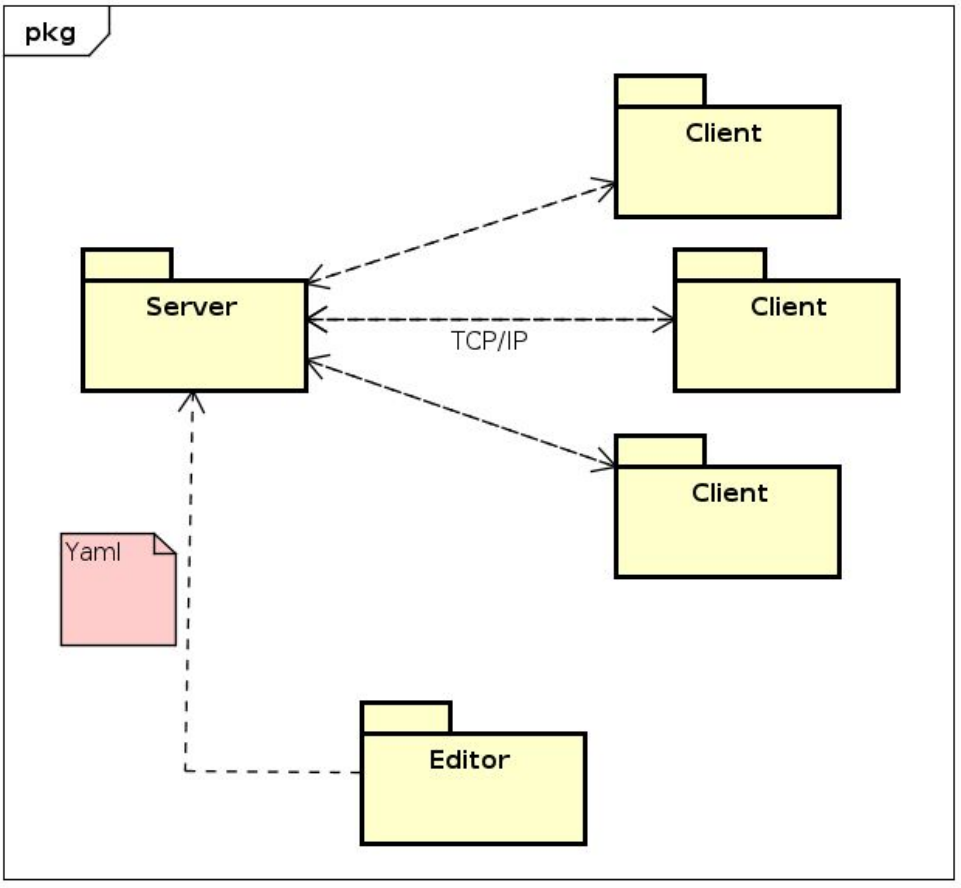
Sistema operativo: Se requiere un sistema operativo que soporte y permita compilar código en lenguaje C++. El programa fue desarrollado y probado en Ubuntu, en su versión 16, utilizando el compilador gcc.

Bibliotecas: Para la compilación y ejecución del programa se requieren las librerías del sistema de threads, matemática y PkgConfig.
Además se requiere tener instalado Gtkmm versión 3.0 y SDL mixer versión 2.

Herramientas: Se requiere de la herramienta Cmake para realizar la compilación.

# 2 Descripción general

El proyecto se divide en tres grandes módulos o aplicaciones, el servidor, el cliente y el editor.



El servidor se encarga de hostear las partidas del juego y la comunicación con los múltiples clientes. Además se encarga de manejar toda la lógica del juego. Se divide en tres módulos, la conexión inicial con el cliente, la fase de juego y el protocolo de comunicación. A su vez, la fase del juego se divide en datos del juego (jugadores, turno, etc) y datos del mundo (mundo, objetos del mundo, viento, etc).

El cliente se encarga de mostrar en una interfaz el estado del juego, además de permitir la interacción con el usuario. Se divide en cinco módulos, la sección del menú principal, la sección de modelo juego, la vista del juego, la comunicación y la sección de sonidos.

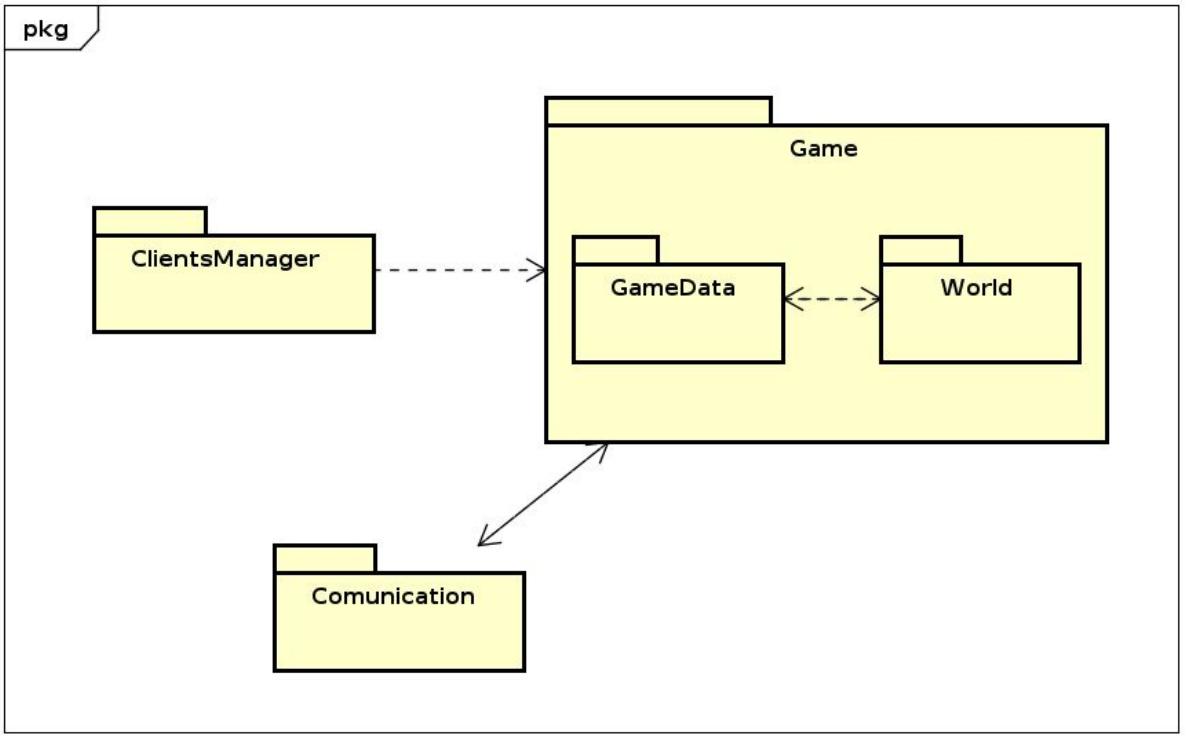
El editor permite al usuario crear sus propios mapas personalizados para luego ser utilizados en el juego, teniendo la posibilidad de poder configurar las posiciones de los worms y de las vigas. También cuenta con la posibilidad de poder configurar cada una de las armas disponibles en el juego y la vida inicial de los worms. El diseño del editor se ha basado en gran parte al patrón MVC. Se divide en tres módulos principales, mapa, armas y vida, y guardado y cargado de mapas.

# 3 Módulos

En esta sección se detallarán los módulos en los que se divide el servidor, el cliente y el editor.

## 3.1 Servidor

Como se explicó anteriormente, el servidor se divide en tres módulos principales:



## 3.1.1 Conexión con el cliente

Este módulo es el encargado de la conexión con el cliente, desde que se conecta a través de la red hasta que se une a una partida.

### 3.1.1.1 Clases

La clase Server es la encargada de aceptar nuevos clientes. Cada vez que se conecta un nuevo cliente crea un Client Handler distinto. Además contiene la lista única de partidas en ejecución.

| **Server** | |
|---|---|
| Atributos: | |
| Socket | Es el socket que permite aceptar nuevos clientes. |
| GamesList | Es la lista de partidas, contiene todas las partidas disponibles. |
| Threads List | Contiene todos los manejadores de clientes que se encuentran activos. |
| Métodos: | |
| run() | Ciclo que acepta clientes hasta que se termine. |
| stop() | Termina el ciclo de aceptación de clientes. |
| check() | Chequea y elimina los manejadores de clientes y partidas inactivos. |

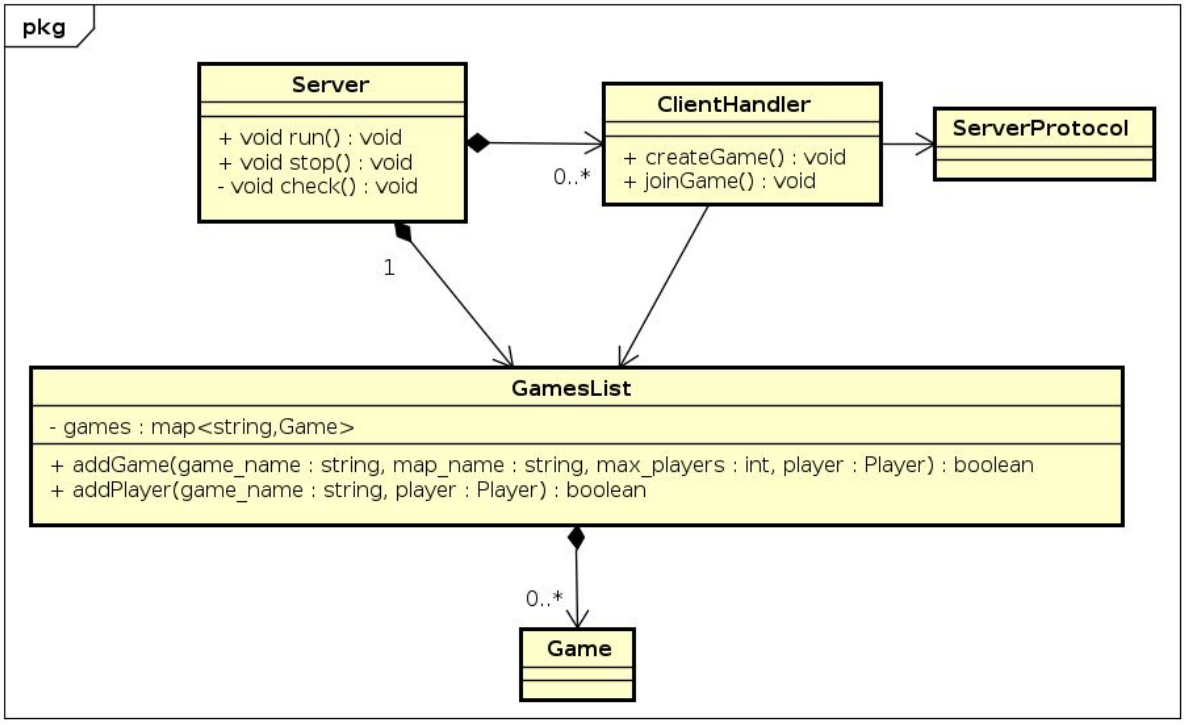
La clase ClientHandler es la encargada de la comunicación con el cliente y la ejecución de la acción solicitada.

| **ClientHandler** | |
|---|---|
| Atributos: | |
| Player | Es el jugador para el cual se creó el handler. |
| GamesList | Es una referencia a la lista de partidas del servidor. |
| Métodos: | |
| run() | Se conecta con el cliente y espera a que éste seleccione una opción. |
| createGame() | Ejecuta las acciones para que el jugador cree una nueva partida. |
| joinGame() | Ejecuta las acciones para que el jugador se una a una nueva existente. |

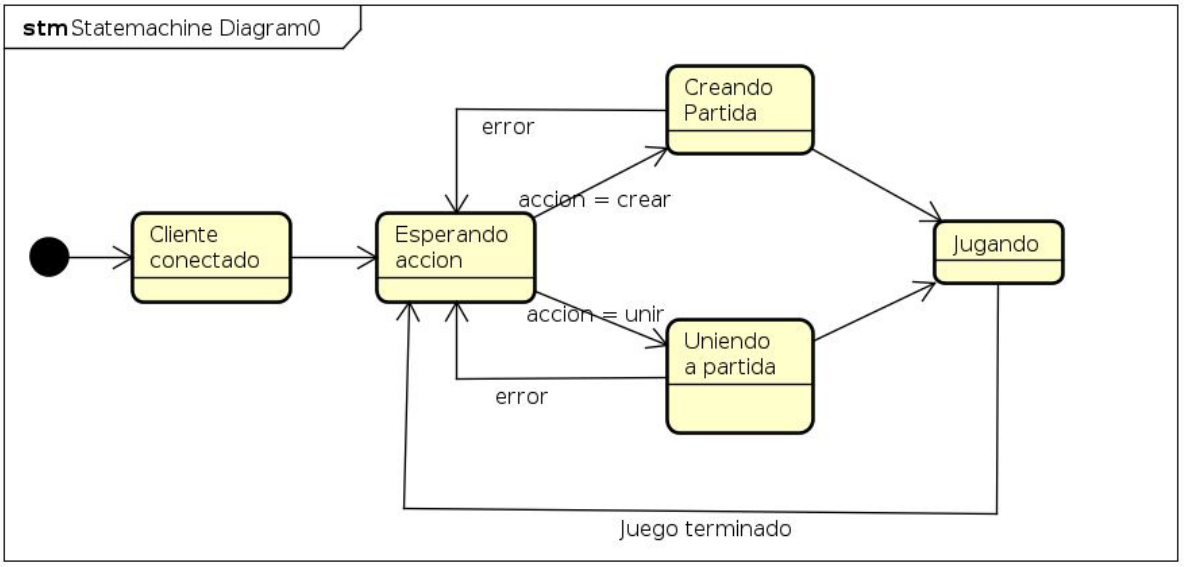
La clase GamesList es la encargada del manejo de las partidas disponibles.

| GamesList | |
|---|---|
| Atributos: | |
| Game List | Es una lista con todas las partidas en ejecución. |
| Métodos: | |
| addGame(game_name, map_name, max_players, player) | Crea una partida con el nombre y mapa indicados. Además agrega al jugador a la misma. |
| addPlayer(game_name, player) | Agrega al jugador a la partida indicada. |

### 3.1.1.2 Diagramas




En este diagrama se puede observar la interacción entre las tres clases principales. El server contiene la lista de partidas y elimina las que ya terminaron. Además crea al client handler cuando se conecta un cliente. Éste se comunica con el cliente a través del protocolo, y cuando corresponda le avisa a la lista de juegos la acción a realizar. La lista de juegos es la encargada de mantener las partidas en ejecución, puede contener muchas partidas distintas.




En este diagrama se muestra todos los estados por los que pasa el cliente, desde que se conecta hasta que se une a la partida. En cuanto se conecta, pasa al estado de selección de acción. Si selecciona crear, pasa al estado de creación de partida. Si selecciona unir, pasa al estado de unir a partida. En ambos casos, si el proceso es exitoso pasa a la fase de juego, caso contrario vuelve al estado de selección de acción. Cuando el juego termina, vuelve al estado de seleccionar acción para que el usuario pueda volver a empezar una partida.

### 3.1.1.3 Protocolo de comunicación

En esta etapa los únicos datos que se envían entre el cliente y el servidor son chars para indicar acciones o resultados, y strings para los nombres.

Primero el servidor recibe un char indicando la acción seleccionada, junto con un string con el nombre del jugador. Luego, el servidor envía al cliente muchos strings, uno por cada mapa disponible (en el caso de crear) y uno por cada partida disponible (en el caso de unir). Luego se recibe del cliente strings conteniendo la selección del usuario. Por último, se envía un carácter indicando si la operación fue exitosa o fallida.

## 3.1.2 Fase de juego - Datos del juego

Este módulo es el encargado del manejo del juego para una partida.

### 3.1.2.1 Clases

La clase Game es la clase principal de la partida. Contiene el ciclo principal, que recibe acciones del jugador que está jugando. Además se encarga de realizar la configuración inicial del juego, como el agregado de los objetos al mapa.

| **Game** | |
|---|---|
| Atributos: | |
| Players List | Es la lista que contiene a todos los jugadores de la partida. |
| Turn | Es el que decide a quién le toca jugar. |
| World | Es el mundo donde se desarrolla el juego. |
| Métodos: | |
| addPlayer(player) | Agrega al jugador a la partida. |
| configure() | Realiza la configuración inicial del juego. |
| run() | Inicia el juego. |

La clase Turn es la clase encargada de seleccionar quien será el próximo jugador que juega.

| **Turn** | |
|---|---|
| Atributos: | |
| Players List | Es una referencia a la lista que contiene a todos los jugadores de la partida. |
| Current Player | Es el jugador que está jugando. |
| Métodos: | |
| beginTurn() | Actualiza el jugador actual. |
| getCurrentPlayer() | Devuelve una referencia al jugador actual. |
| gameEnded() | Determina si el juego terminó o no, en función de la cantidad de jugadores vivos. |

La clase Player es la que contiene los datos del jugador, en particular el nombre y la lista de worms que le pertenecen.

| **Player** | |
|---|---|
| Atributos: | |
| Name | Es el nombre del jugador. |
| WormsList | Es la lista de worms del jugador. |
| WeaponsList | Es la lista de armas y munición del jugador. |
| Protocol | Es el protocolo de comunicación con el jugador. |
| Métodos: | |
| beginTurn() | Actualiza el worm actual del jugador. |
| getCurrentWorm() | Devuelve una referencia al worm actual del jugador. |

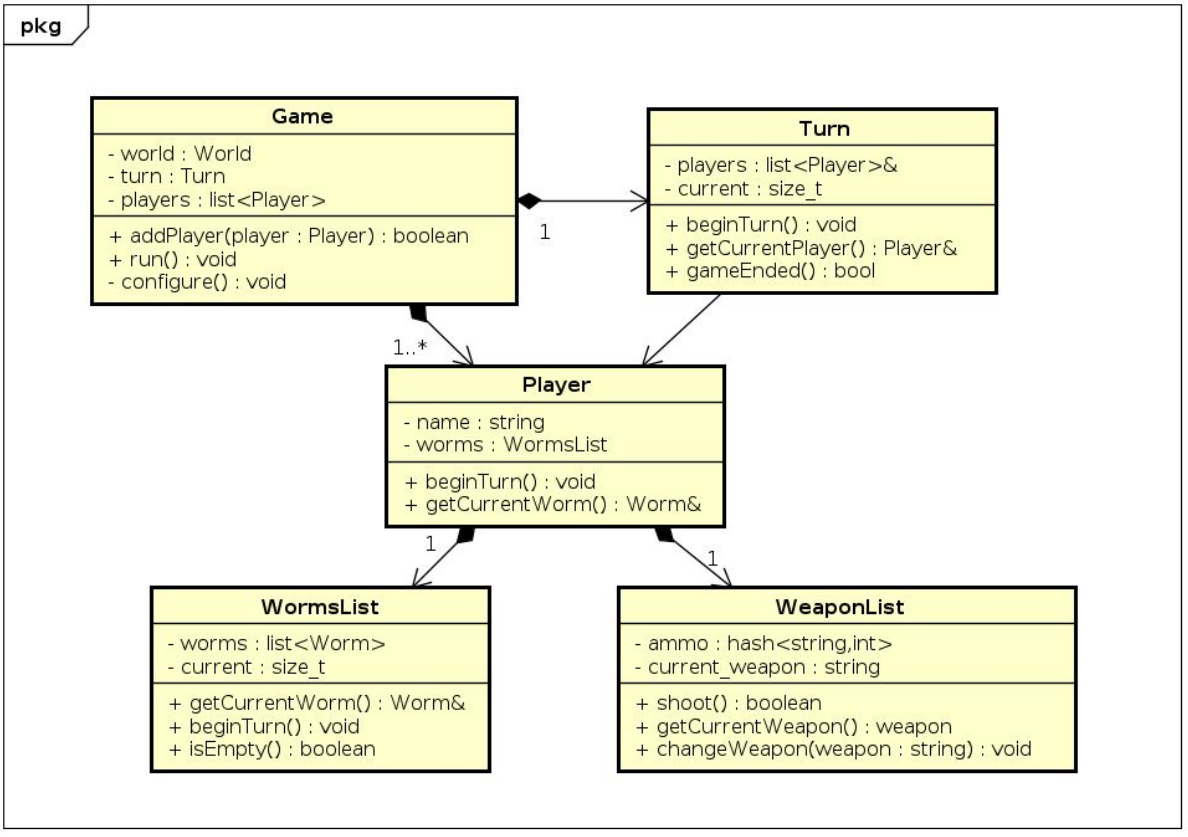
La clase WormsList es la que contiene los worms del jugador, y es la que determina cual es el worm con el que juega.

| **WormsList** | |
|---|---|
| Atributos: | |
| Worms List | Es la lista de worms del jugador. |
| Current Worm | Es el gusano actual. |
| Métodos: | |
| beginTurn() | Actualiza el worm actual. |
| getCurrentWorm( ) | Devuelve una referencia al worm actual. |
| isEmpty() | Determina si aun quedan worms vivos. |

La clase WeaponList es la que contiene las armas del jugador con su respectiva munición, y es la que determina si puede disparar o no..

| **WeaponList** | |
|---|---|
| Atributos: | |
| Ammo hash | Es la munición actual de cada arma. |
| Current weapon | Es el arma actual. |
| Métodos: | |
| changeWeapon(weapon) | Actualiza el arma actual. |
| getCurrentWeapon() | Devuelve el arma actual. |
| shoot() | Determina si puede disparar o no, y disminuye la munición. |

## 3.1.2.2 Diagramas



En este diagrama se puede observar como la clase juego contiene todos los datos del mismo, como el turno y el mundo. A su vez, el turno actúa sobre los jugadores para seleccionar a actual, y cada jugador contiene su lista de gusanos y lista de armas.

### 3.1.2.3 Archivo de configuración

El juego se configura en función de dos archivos de configuración. Por un lado, el archivo de configuración general del servidor, que es igual para todas las partidas, permite configurar la velocidad de los gusanos, la velocidad de disparo de las armas, el daño y radio de explosión de todas las armas, la velocidad mínima y máxima del viento, la gravedad, la duración del turno, etc.

Por otro lado, el archivo de configuración que provee el editor, determina la posición inicial de los gusanos y vigas, la rotación y longitud de las vigas, la vida inicial de los gusanos y la munición inicial de cada arma. Este archivo es único por cada escenario creado, y lo elige el jugador que crea la partida.

### 3.1.2.4 Protocolo de comunicación

Esto se explicará en el módulo del protocolo de comunicación.

## 3.1.3 Fase de juego - Datos del mundo

Este módulo es el encargado del manejo del mundo de la partida. Contiene todos los objetos agregados al mundo como vigas, armas y gusanos.

### 3.1.3.1 Clases

La clase World es la clase encargada de la simulación del mundo físico para el juego. Además, es la que maneja el ciclo de vida de un objeto en el mundo.

| **World** | |
|---|---|
| Atributos: | |
| b2world | Es el mundo del juego, proporcionado por la librería Box2D. |
| PhysicalObject List | Es la lista de objetos que hay en el mundo. |
| Wind | Es el viento que afecta a los objetos del mundo. |
| Métodos: | |
| addObject(object, position) | Agrega el objeto al mundo en la posición indicada. |
| removeObject(object) | Elimina el objeto del mundo. |
| run() | Contiene el ciclo que se encarga de actualizar el mundo y hacer que los objetos interactúen. |
| getClosestObject(position, end) | Busca y devuelve el objeto más cercano a position, en la dirección de end. |

La clase PhysicalObject es una clase abstracta que contiene el comportamiento común para todos los objetos, ya sean gusanos, vigas o armas.

| **PhysicalObject** | |
|---|---|
| Atributos: | |
| b2body | Es la representación del objeto en el mundo. |
| bool is_dead | Determina si el objeto está muerto o no. |
| Métodos: | |
| initializeBody() | Inicializa los datos del objeto, como su tamaño, masa, etc. |
| destroyBody() | Destruye el cuerpo del objeto. |
| collide_with_something(other) | Actúa cuando el objeto colisiona con otro. |
| getPosition() | Devuelve la posición actual del objeto. |
| isMoving() | Determina si el objeto se está moviendo o no. |
| isDead() | Determina si el objeto está muerto o no. |

La clase Worm modela a un gusano en el mundo. Contiene todos los datos del mismo como su vida o su arma actual.

| **Worm (PhysicalObject)** | |
|---|---|
| Atributos: | |
| Life | Es la vida del gusano. |
| Dir | Es la dirección actual del gusano (izquierda o derecha). |
| Weapon | Es el arma actual del gusano. |
| Métodos: | |
| move() | Mueve al gusano o lo hace saltar. |
| shoot() | Dispara el arma del gusano. |
| collide_with_something(other) | Actúa cuando el gusano colisiona con otro objeto. |
| receiveWeaponDamage(damage, epicenter) | Recibe daño del arma, en función de la distancia entre ellos. |

La clase Weapon modela a el comportamiento de todas las armas o herramientas. En general todas las armas se comportan igual, excepto algunos casos especiales como la teletransportación o el bate.

| **Weapon (PhysicalObject)** | |
|---|---|
| Atributos: | |
| Damage | Es el daño máximo que produce el arma. |
| Radius | Es el radio de la explosión del arma. |
| Métodos: | |
| shoot() | Dispara el arma. |
| explode() | Explota el arma. |
| collide_with_something(other) | Explota el arma si no es con cuenta regresiva. |

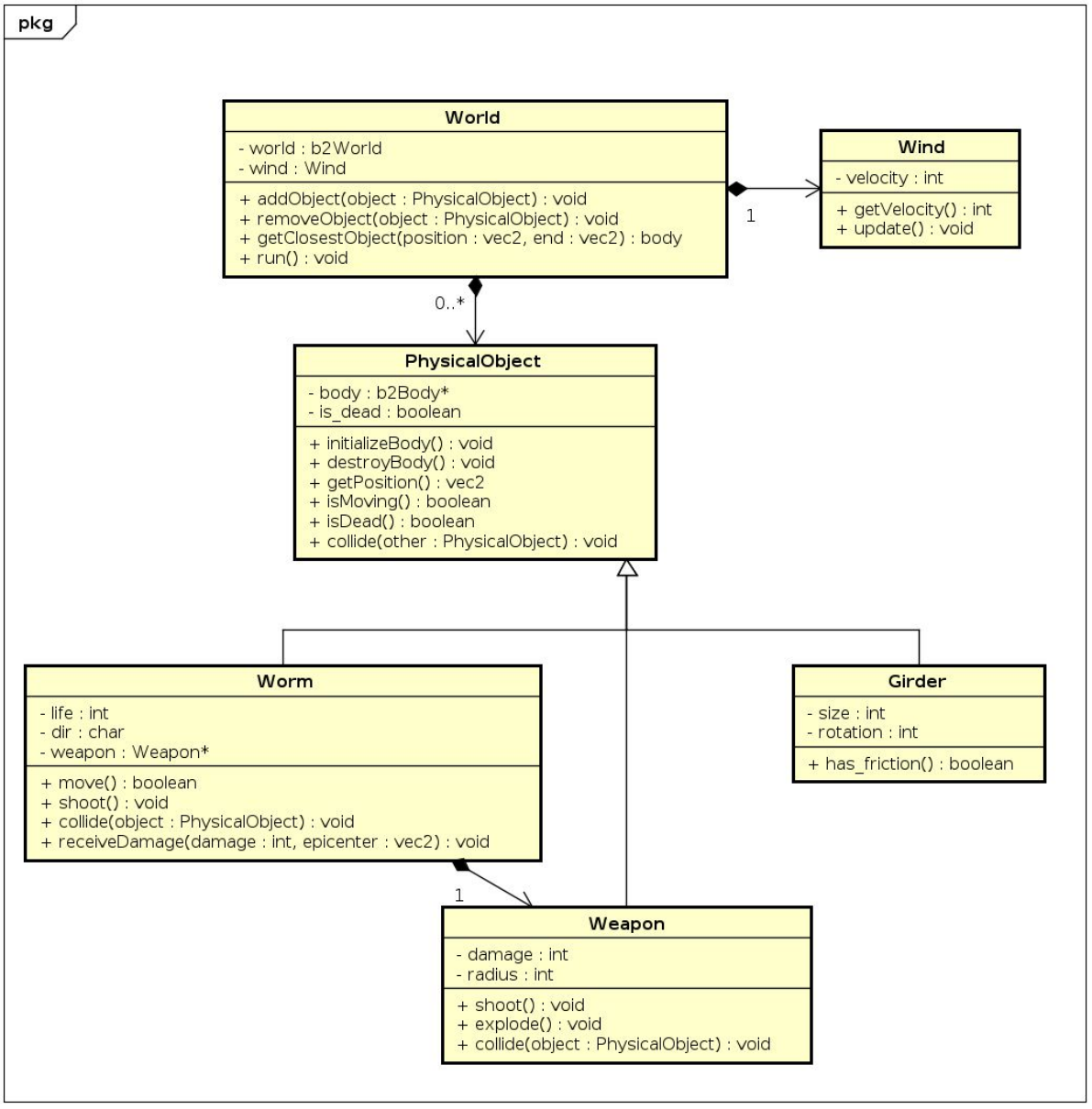
Los casos especiales de armas son:

- Fragmentables: Al explotar lanzan varios fragmentos al aire.
- Bate: No hace daño en un radio, sino solo en una dirección.
- Dinamita: No sale lanzada con una velocidad, se lanza en el lugar.
- Teletransportación: Teletransporta al gusano.
- Ataque Aéreo: Lanza misiles desde el cielo.
- Banana: Rebota varias veces antes de explotar.

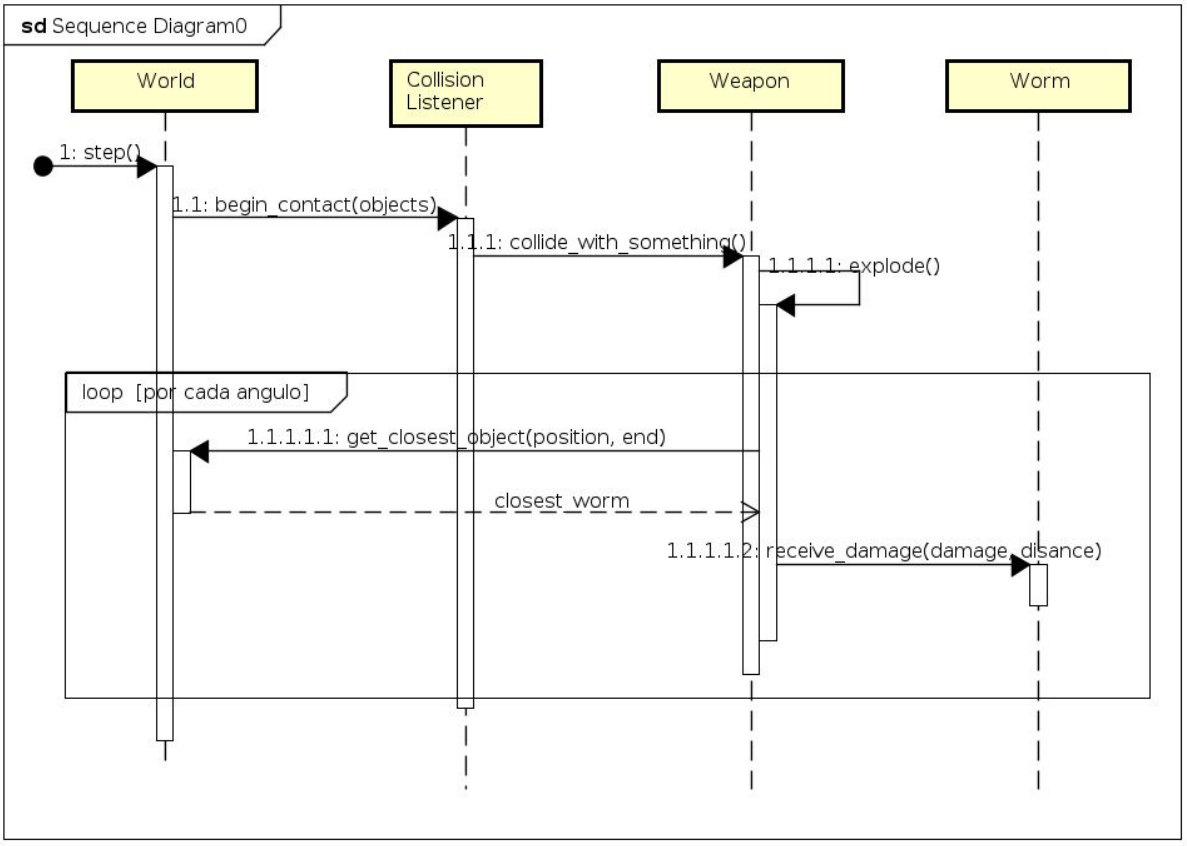
La clase Girder modela a una viga en el mundo.

| **Girder (PhysicalObject)** | |
|---|---|
| Atributos: | |
| Size | Es el tamaño de la viga. |
| Rotation | Es el ángulo de rotación de la viga. |
| Métodos: | |
| has_friction() | Determina si la viga tiene fricción o no, en función de su rotación. |

### 3.1.3.2 Diagramas




En este diagrama se puede observar como la clase world contiene physical objects, los cuales pueden ser gusanos, armas o vigas. Además contiene el viento, que afecta a los objetos. Por último, el gusano interactúa con el arma, para poder disparar.




En este diagrama se muestra la secuencia de explosión de una arma normal, desde que colisiona con algo hasta que hace daño a los gusanos.

## 3.1.4 Protocolo de comunicación

Este módulo es el encargado de la comunicación con los jugadores durante el juego. Por un lado recibe las acciones a realizar del jugador actual, por otro lado envía los datos del juego a todos los jugadores.

### 3.1.4.1 Clases

La clase ServerProtocol es la encargada de crear los mensajes que luego se enviarán, además de recibir los mensajes del cliente.

| **ServerProtocol (Protocol)** | |
|---|---|
| Atributos: | |
| Socket | Es el socket que permite comunicarse con el cliente. |
| Métodos: | |
| receive() | Recibe un mensaje y ejecuta la acción correspondiente. |
| sendObject(object) | Envía todos los datos importantes del objeto. El objeto puede ser un gusano o un arma. |
| sendStartTurn(player_id, worm_id) | Envía la señal de inicio de turno, junto con el jugador y el gusano que le toca jugar. |
| sendEndTurn() | Envía la señal de que termino el turno del jugador actual. |
| sendEndGame(winner) | Envía la señal de fin del juego, junto con el nombre del ganador. |
| Otros métodos | Hay métodos para enviar la configuración inicial, como las vigas y los jugadores presentes. Además hay métodos especiales para cada mensaje que requiere el cliente, como cambio de arma o actualización de la mira de disparo. |

La clase DataSender es la encargada de enviar los mensajes a todos los jugadores. Por un lado está constantemente enviando la información de los objetos. Por otro lado, envía mensajes especiales cuando se le solicita.

| **DataSender** | |
|---|---|
| Atributos: | |
| PlayerDataSender list | Lista de los senders de los clientes. |
| PhysicalObject list | Referencia a la lista de objetos que hay en el mundo. |
| Métodos: | |
| run() | Ciclo que envía constantemente los datos de los objetos, como por ejemplo las posiciones. |
| Otros métodos | Hay métodos para enviar la configuración inicial, como las vigas y los jugadores presentes. Además hay métodos especiales para cada mensaje que requiere el cliente, como cambio de arma o actualización de la mira de disparo. |

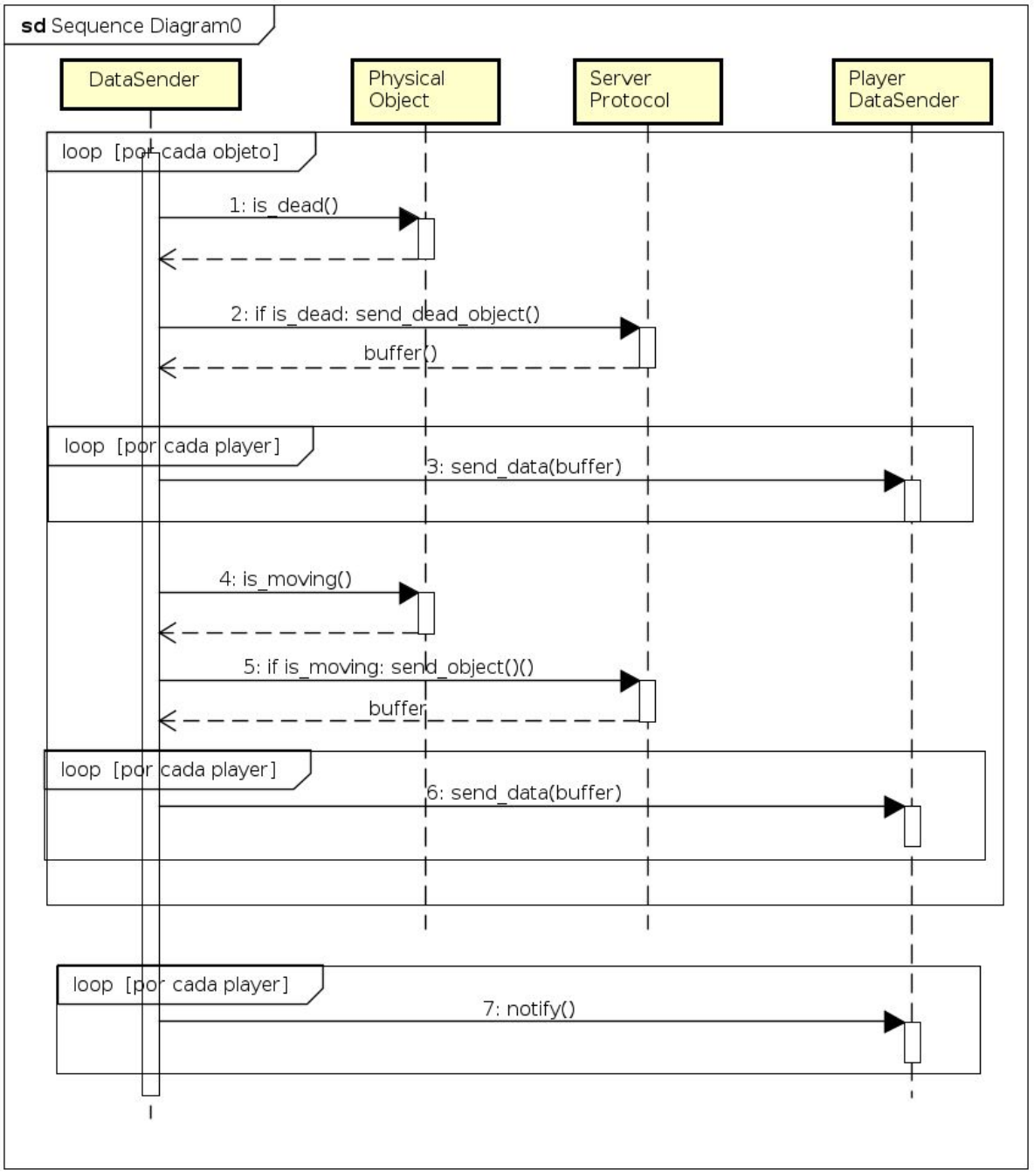
La clase PlayerDataSender es la encargada de enviar los mensajes a un jugador en particular. Se comporta como una cola bloqueante, es decir, solo envía datos cuando se le notifica que se agregaron nuevos.

| **PlayerDataSender** | |
|---|---|
| Atributos: | |
| Player | Jugador conectado al sender. |
| Métodos: | |
| sendData(data) | Se agrega el mensaje a la cola de mensajes a enviar. |
| notify() | Se le notifica que se agregaron nuevos datos. |
| run() | Ciclo que envía datos siempre que la cola no esté vacía y se le haya notificado. |

La clase PlayerDataReceiver es la encargada de recibir los mensajes de un jugador en particular. Cada vez que recibe un mensaje, realiza la acción indicada siempre y cuando sea el turno de ese jugador.

| **PlayerDataReceiver** | |
|---|---|
| Atributos: | |
| Player | Jugador conectado al receiver. |
| is_my_turn | Variable que indica si es el turno del jugador o no |
| Métodos: | |
| beginTurn() | Indica que empezó el turno del jugador |
| endTurn() | Indica que finalizó el turno del jugador |
| analizeReceivedData() | Realiza la acción indicada por el mensaje del cliente. |

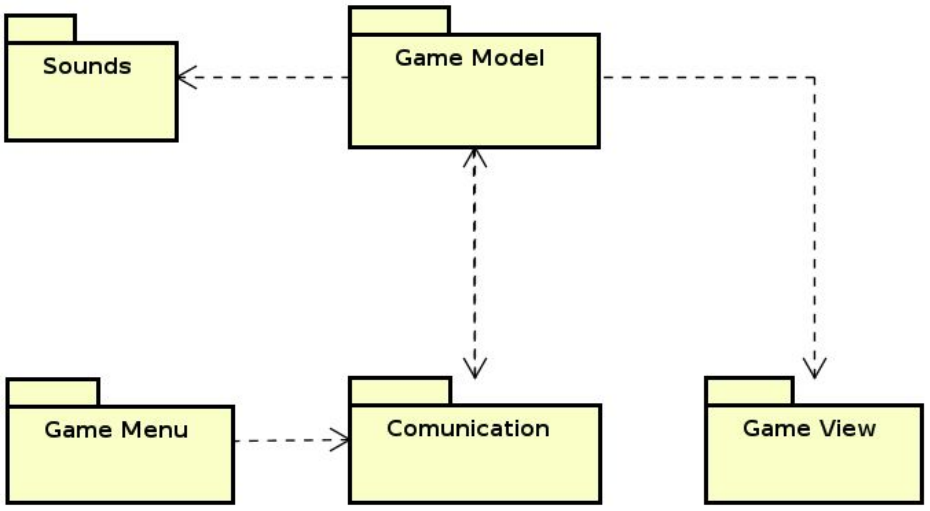
### 3.1.4.2 Diagramas




En este diagrama se observa como el DataSender se encarga de recorrer los objetos del mundo y pedirle al protocolo la información para enviar, dependiendo si el objeto está muerto o no. Luego agrega el mensaje en la cola de cada jugador. Por último notifica a los jugadores para que envíen los mensajes.

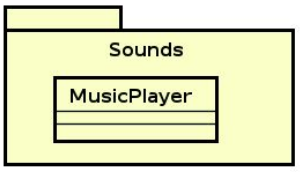
# 3.2 Cliente

El cliente se divide en cinco módulos:




## 3.2.1 Sonidos

Este módulo es el encargado de la reproducción de sonidos y música en el cliente. El módulo de sonidos tiene la siguiente organización:



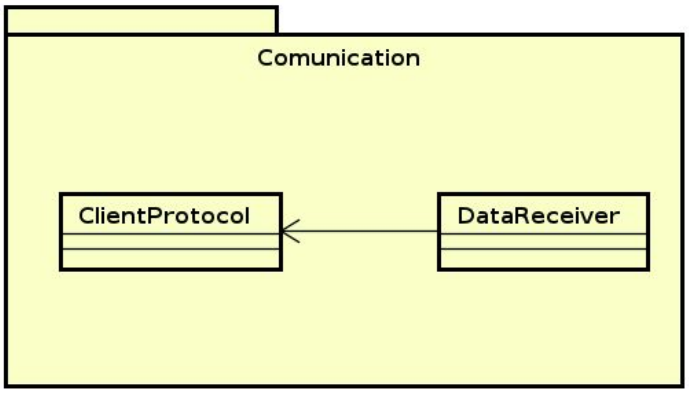
#### 3.2.1.1 Clases

La clase MusicPlayer es la encargada de cargar SDL y SDL_MIXER que permiten reproducir música y sonidos a partir del path de un archivo de música. En este módulo se definió también una excepción MusicPlayerException la cual es lanzada cuando el MusicPlayer no puede iniciar SDL o SDL_MIXER.

| **MusicPlayer** | |
|---|---|
| Atributos: | |
| Mix_Music* music | Es la música de fondo del juego. |
| std::map<int, Mix_Chunk*> effects | Hash que almacena los sonidos que se van reproduciendo en el juego. |
| Métodos: | |
| void check(int channel) | Recorre el hash effects y libera todos los sonidos que han terminado de ejecutarse. Además libera el audio almacenado en el canal channel. |
| void addEffect(const std::string& audio) | Agrega un nuevo sonido al hash y lo reproduce. |
| void playMusic() | Reproduce la música de fondo |
| stop() | Detiene la reproducción de la música de fondo |

## 3.2.2 Comunicación

Este módulo es el encargado de la comunicación entre el cliente y el servidor. El módulo de comunicación tiene la siguiente organización:




### 3.2.2.1 Clases

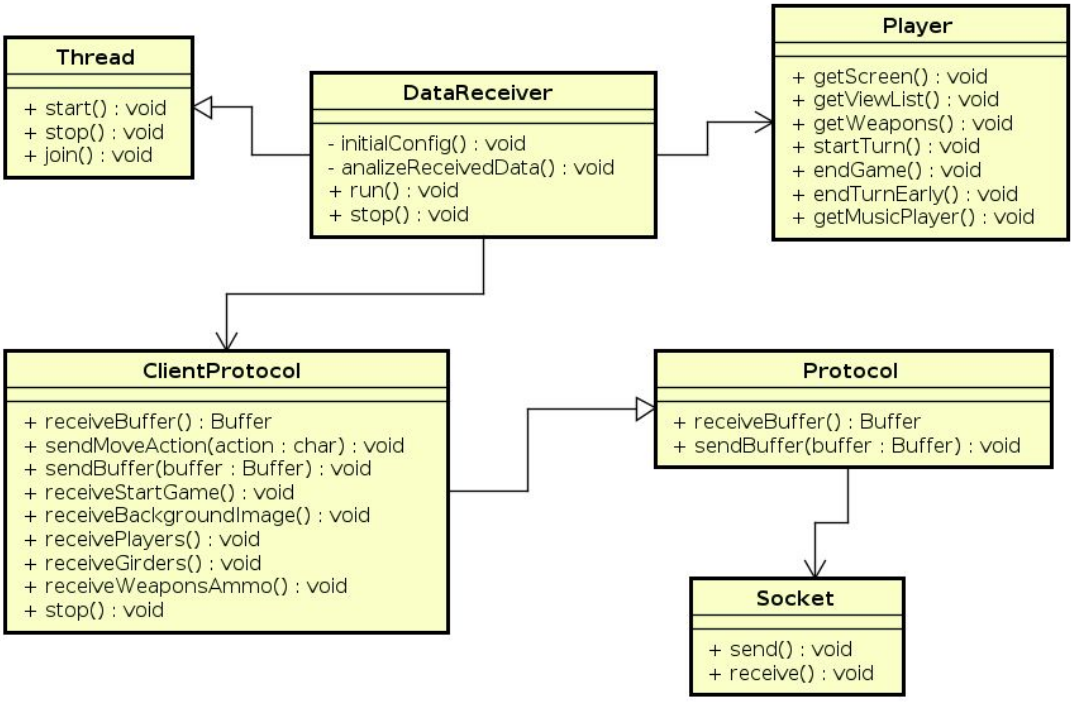
La clase ClientProtocol es la que se encarga de enviar y recibir datos del servidor con un formato determinado.

| **ClientProtocol : Protocol** | |
|---|---|
| Atributos: | |
| Socket | Socket de comunicación con el servidor. |
| Métodos: | |
| sendEndTurn() | Envía un mensaje de finalización de turno |
| receivePlayers(PlayersList& player_list) | Recibe los jugadores de la partida |
| sendMoveAction(char action) | Envía un mensaje indicando que el worm se movió |
| sendWeaponShoot(int32_t angle, int32_t power, int32_t time) | Envía un mensaje de acción de disparo, indicando el ángulo, la potencia y el tiempo de explosión |

La clase DataReceiver es un thread que se encarga de ir recibiendo los mensajes enviados por el servidor y además se encarga de parsear y analizar el mensaje recibido.

| **DataReceiver : Thread** | |
|---|---|
| Atributos: | |
| Player& player | El jugador |
| ClientProtocol& protocol | Protocolo que se encarga de comunicarse con el servidor |
| Métodos: | |
| initialConfig() | Recibe los datos de configuración inicial |
| analizeReceivedData(Buffer buffer) | Analiza los datos recibidos y realiza las acciones correspondientes |
| run() | Comienza a recibir los datos del servidor |
| stop() | Detiene el proceso de recibir mensajes |

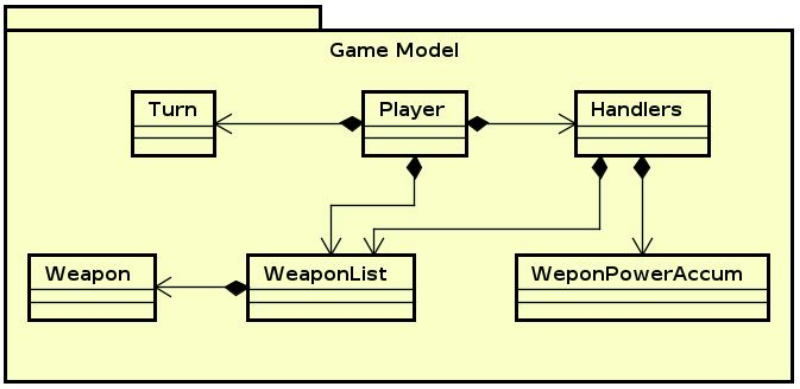
## 3.2.2.2 Diagramas




En este diagrama se puede observar como la clase DataReceiver se relaciona con el protocolo y el Player. El receiver recibe y parsea el mensaje del servidor y luego envía la acción a la clase correspondiente.

## 3.2.3 Modelo del juego

Este módulo es el encargado de la modelización del juego. Este módulo tiene la siguiente organización:




### 3.2.3.1 Clases

La clase Player es la clase principal de este módulo, la cual se encarga de representar al Jugador.

| **Player** | |
|---|---|
| Atributos: | |
| ClientProtocol | Protocolo del cliente |
| std::string name | Nombre del jugador |
| WeaponList weapons | Lista de las armas del jugador |
| ViewsList | Lista de los objetos visibles en el mapa |
| Métodos: | |
| startTurn(int worm_id, int player_id, float wind) | Comienza el turno del jugador con el id player_id, el cual podrá controlar al worm con id worm_id y el viento de ese turno es wind. Si es su turno se activaran los handlers de ese jugador |
| endGame(const std::string& winner) | Finaliza el juego e indica quién fue el ganador |
| chageWeapon(std::string weapon) | Cambia el arma actual por la indicada |
| shoot(Position position) | Realiza el disparo de un arma teledirigida en la posición especificada |
| shoot(int angle, int power, int time) | Realiza el disparo de un arma con el ángulo, potencia y tiempo especificados |

La clase WeaponPowerAccum es la clase que se encarga de obtener la potencia del arma, según cuanto tiempo el jugador mantuvo presionado el botón.

| **WeaponPowerAccum** | |
|---|---|
| Atributos: | |
| int actual_time | Tiempo actual |
| int max_time | Tiempo máximo |
| int power | Potencia del tiro |
| Métodos: | |
| start() | Comienza a acumular la potencia y cuando alcanza la potencia máxima realiza el disparo |
| stop() | Detiene el contador de potencia y realiza el disparo con la potencia actual |

La clase Turn es la clase que se encarga de contar el tiempo del turno para mostrarlo al jugador..

| **Turn** | |
|---|---|
| Atributos: | |
| int actual_time | Tiempo actual |
| TurnLabel& time_label | Label que indica el tiempo faltante para la finalización del turno |
| sigc::connection my_connection | |
| Métodos: | |
| start() | Comienza la cuenta regresiva del turno |
| reduceTime() | Reduce el tiempo restante del turno a 3 segundo |
| stop() | Detiene el contador y finaliza el turno |

La clase Handlers se encarga de definir las acciones u operaciones que se deben realizar al producirse eventos específicos. En este caso se define qué se debe hacer al apretar las teclas enter, backspace, barra espaciadora, las flechas, los números 1, 2 , 3, 4 y 5, o si se oprime el click izquierdo del mouse.

La clase ScrollHandler se encarga de definir la acción que se debe realizar cuando el mouse se encuentra en una posición determinada. En este caso, si el mouse se encuentra en los bordes del mapa, se moverá la vista hacia esa dirección. Es decir, si el mouse se encuentra en el extremo

derecho, entonces la vista comenzará a moverse hacia la derecha hasta que se alcance el final del mapa o se saque el mouse del extremo derecho.

La clase Weapon es la clase principal de las armas del juego. Todas las armas heredarán de esta clase.

| **Weapon** | |
|---|---|
| Atributos: | |
| std::string name | Nombre del arma |
| unsigned int ammo | Munición del arma |
| bool has_Scope | Indica si el arma tiene mira o no |
| bool is_Timed | Indica si el arma tiene cuenta regresiva |
| Métodos: | |
| shoot() | Disminuye en uno la munición del arma |
| hasAmmo() | Indica si el arma tiene munición |
| hasScope() | Indica si el arma tiene mira |
| isTimed() | Indica si el arma tiene cuenta regresiva |

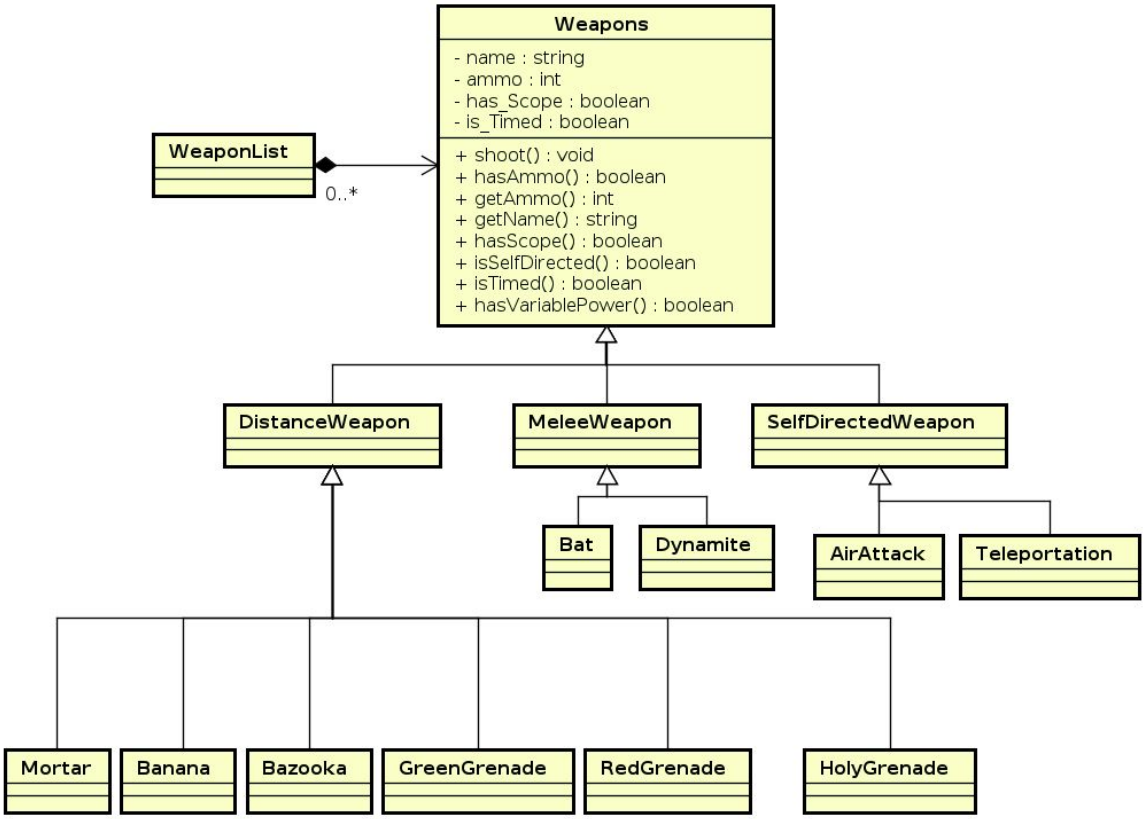
La clase WeaponList es la clase que almacenará todas las armas del juego y sabrá cuál es el arma actual.

| **WeaponList** | |
|---|---|
| Atributos: | |
| std::map<std::string, weapon_ptr> weapons | Hash que contiene todas las armas |
| std::string current_weapon | Arma actual del jugador |
| Métodos: | |
| add(std::string weapon, int ammo) | Crea el arma pasada y la agrega a la lista |
| changeWeapon(std::string weapon) | Cambia el arma actual por la pasada |
| getCurrentWeapon() | Devuelve una referencia al arma actual |

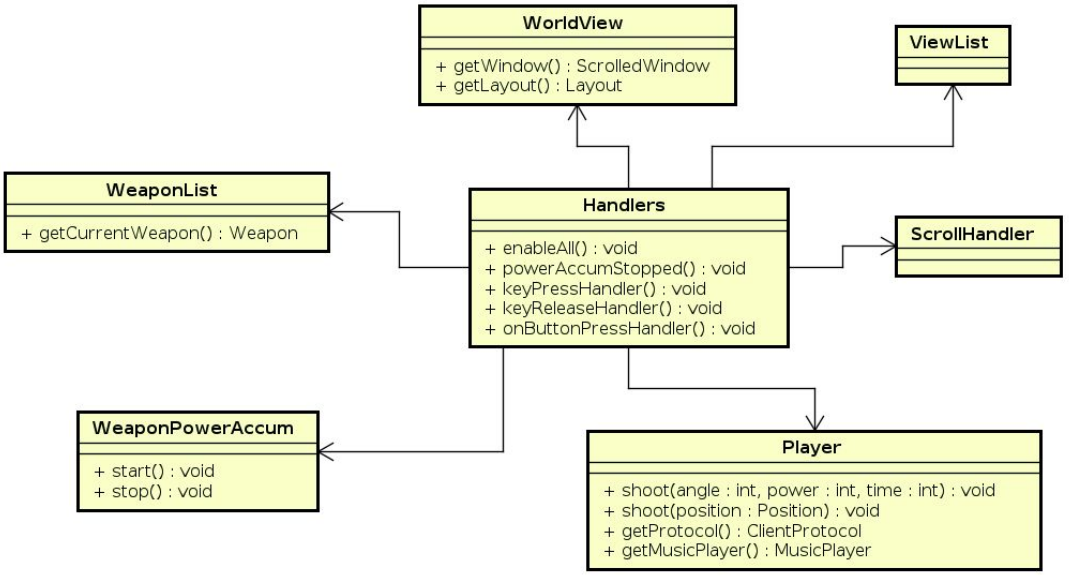
Las clases MeleeWeapon, DistanceWeapon y SelfDirectedWeapon heredan de la clase Weapon y se encargan de representar las armas que son de cuerpo a cuerpo, las armas de distancia y las armas teledirigidas. Luego, cada arma del juego tiene su propia clase, las cuales heredan de alguna de estas tres clases.

### 3.2.3.2 Diagramas

En este diagrama se puede observar la jerarquía entre las distintas clases de armas.

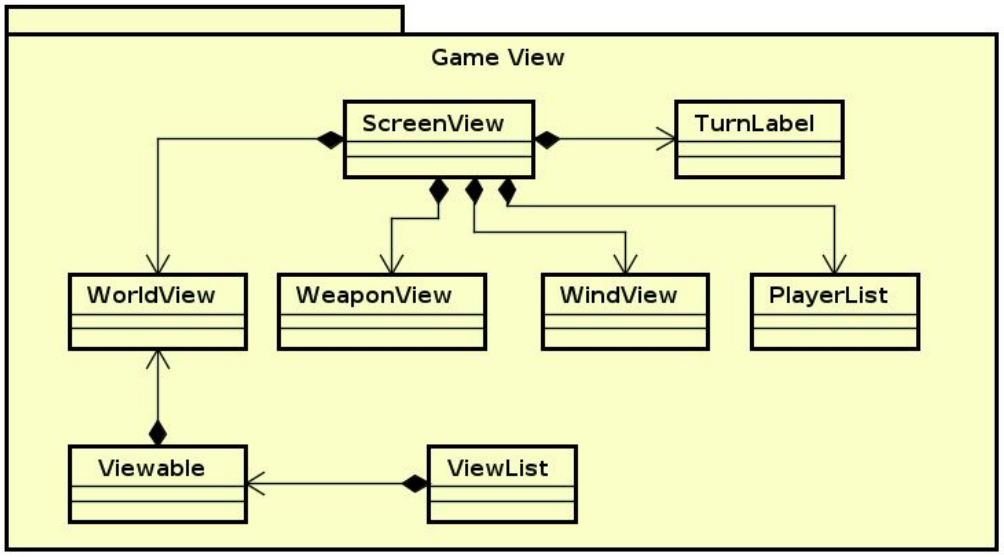



En este diagrama se puede observar cómo la clase Handlers, que recibe la acción del usuario, envía el mensaje a la clase correspondiente según cual sea la acción.



## 3.2.4 Vista del juego

Este módulo es el encargado de la parte visual del juego. Este módulo tiene la siguiente organización:




### 3.2.4.1 Clases

La clase WorldView es la clase más importante pues es la clase que representa el mundo del juego, el cual se agregaran las vistas de los objetos en posiciones específicas del mundo y se los podrá mover o remover de éste.

| **WorldView** | |
|---|---|
| Atributos: | |
| Gtk::Layout world | Contenedor de los objetos |
| Water water | Vista del agua |
| Métodos: | |
| setBackgroundImage(image) | Setea la imagen de fondo |
| addElement(element, position, width, height, focus) | Agrega al elemento a la posición indicada, y hace focus en él si el parámetro focus lo indica |
| moveElement(element, position, width, height, focus) | Mueve el elemento a la posición indicada, y hace focus en él si el parámetro focus lo indica |
| removeElement(element) | Elimina al elemento de la vista |
| setFocus(element) | Hace focus al elemento |

La clase ViewsList es la clase que se encarga de almacenar y controlar los objetos del mapa.

| **ViewsList** | |
|---|---|
| Atributos: | |
| WorldView& world | Mundo en donde se almacenan las vistas de los objetos |
| std::unordered_map<int, WormView> worms | Lista con las vistas de los worms del juego |
| std::unordered_map<int, BulletView> weapons | Lista con las vistas de las balas del juego |
| std::vector<GirderView> girders | Lista con las vistas de las vigas del juego |
| int current_worm_id | Id del worm actual |
| int weapon_focused | Id de la bala que tiene focus |
| int worm_focused | Id del worm que tiene focus |
| Métodos: | |
| updateWormData(id, player_id, pos_x, pos_y, life, dir, colliding) | Actualiza la posición y vida del worm con el id correspondiente |
| updateWeaponData(id, weapon_name, pos_x, pos_y) | Actualiza la posición de la bala del arma |
| changeWeapon(weapon_name) | Actualiza la vista del worm para que tenga como arma la pasada |
| updateScope(angle) | Actualiza la posición de la mira |
| setVictory() | Actualiza la vista de los worms por la animación de los worms festejando |

La clase Viewable es la clase de la cual heredan los objetos visibles en el mapa (worms, vigas y balas) y se encarga de controlar la visualización de estos objetos.

| **Viewable** | |
|---|---|
| Atributos: | |
| WorldView& world | Mundo en donde se agrega el objeto Viewable |
| bool has_focus | Indica si el objeto puede tener focus o no |
| Métodos: | |
| addToWorld(position, width, height) | Agrega el objeto Viewable al world |
| move(position, width, height) | Mueve el objeto a la posición indicada |
| remove() | Elimina el objeto del mundo |
| setFocus(focus) | Establece si el objeto puede tener focus o no |

Las clases GirderView, BulletView y WormView heredan de la clase Viewable. Las dos primeras no tienen una gran complejidad. La clase GirderView tiene la imagen, la rotación y el tamaño de la viga. La clase BulletView tiene la imagen y el nombre del arma. Por otro lado, la clase WormView es más compleja que las demás debido a la cantidad de acciones que puede ejecutar el worm que impactan en la vista.

| **WormView** | |
|---|---|
| Atributos: | |
| int life | Vida actual del worm |
| char dir | Dirección actual del worm |
| bool is_moving | Indica si se está moviendo |
| std::string weapon | Arma actual del worm |
| Gtk::Image image | Imagen del worm |
| Métodos: | |
| changeWeapon(weapon) | Actualiza el arma actual por la pasada, actualiza el vector scope_vector con las imágenes de la nueva arma y cambia la imagen del worm por la imagen del worm con la nueva arma |
| updateData(life, dir, position, colliding, isCurrentWorm, hasShoot) | Actualiza la posición del worm y la vista del worm (la imagen y su vida) |
| setNewImage() | Método que cambia la imagen del worm según si se está moviendo, está por disparar, etc. |

La clase TurnLabel se encarga de controlar un label que indica de quién es el turno y cuánto tiempo restante queda para que el turno finalice (si es el turno del jugador). También se utiliza para indicar si el juego finalizó y para indicar quién fue el ganador.

La clase PlayersList se encarga de mantener los labels de todos los jugadores imprimiendo la información de estos en la vista (el id del jugador, su nombre y su vida).

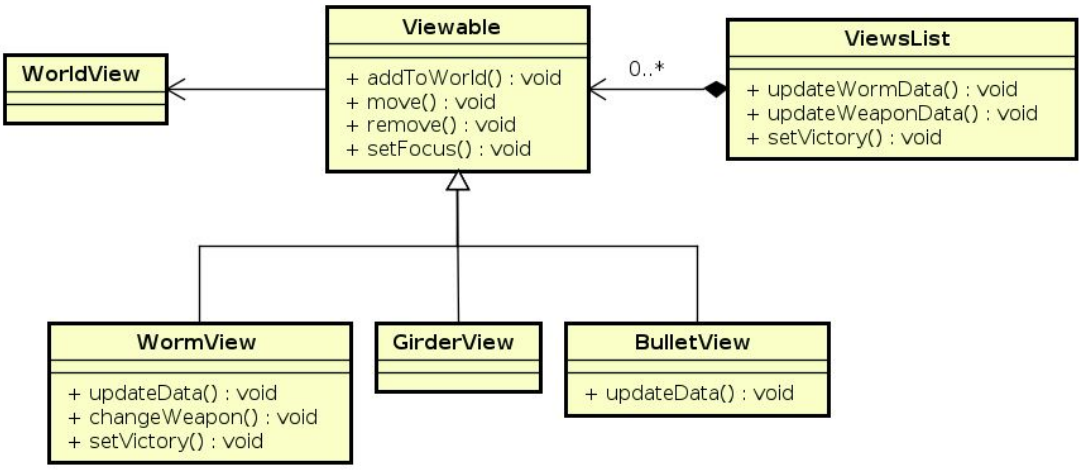
La clase WindView se encarga de mostrar las características del viento en el turno actual. Muestra un label que indica la magnitud del viento y una imagen que indica el sentido del viento.

La clase WeaponView se encarga de mostrar los datos necesarios de las armas del jugador (Imagen del arma, nombre y munición). Esta vista está compuesta por botones, los cuales al ser seleccionados realizan el cambio del arma actual por el arma seleccionada.

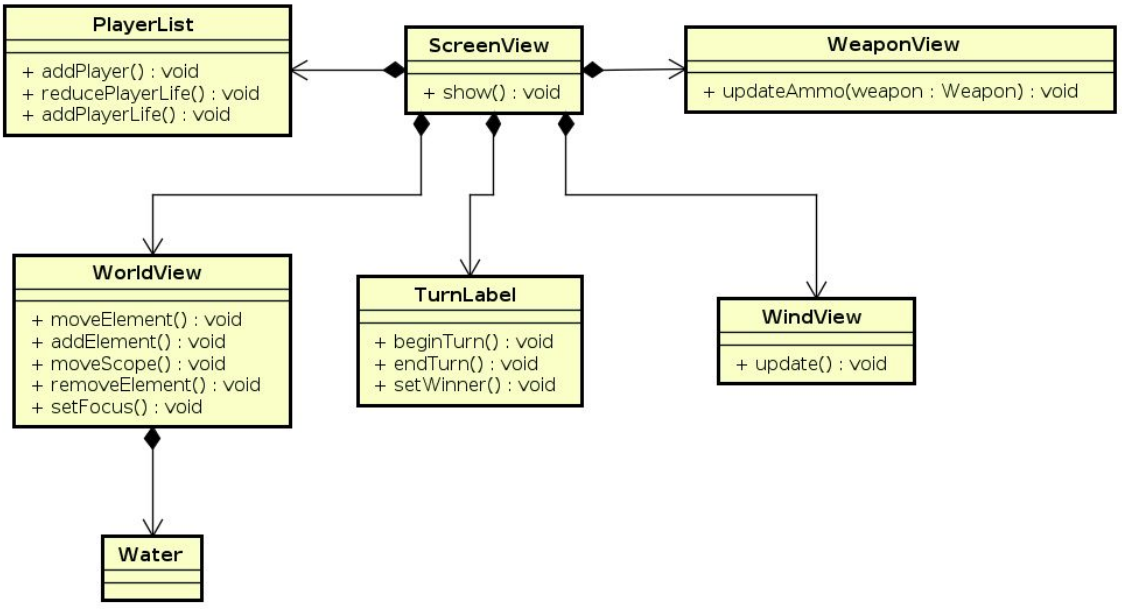
| **WeaponView** | |
|---|---|
| Atributos: | |
| WeaponList& weapons | Lista de las armas del jugador |
| unordered_map<string, WeaponButton> | Lista con los botones de la vista |
| Métodos: | |
| updateAmmo(weapon) | Actualiza el label de la munición del botón del arma correspondiente |

La clase ScreenView es el contenedor principal que almacena a los demás contenedores (WeaponView, WorldView, TurnLabel, PlayersList y WindView) y los muestra.

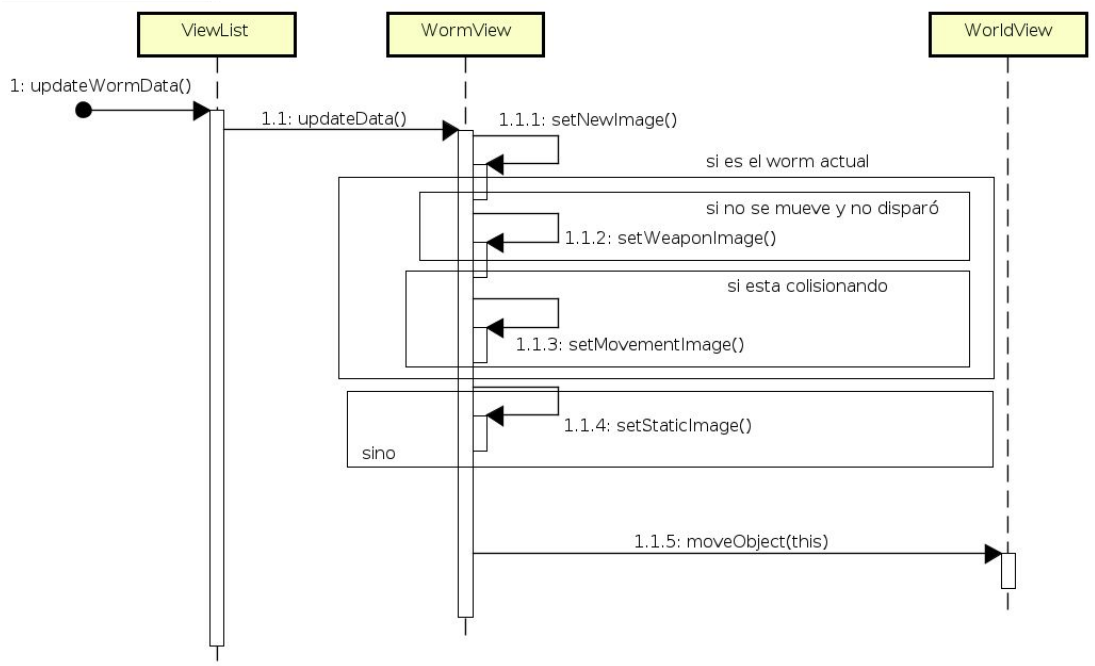
### 3.2.4.2 Diagramas




En este diagrama se puede observar como las clases de las vistas de los objetos se relacionan entre sí.



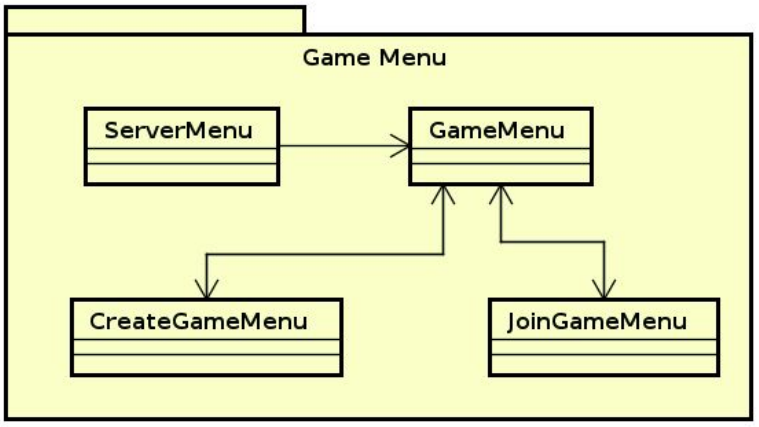

En este diagrama se puede observar como la clase ScreenView almacena las distintas partes visuales del juego.




En este diagrama se puede observar cómo se modifica la vista del worm al recibir la señal updateWormData, en funcion de los datos que recibe del servidor. Esto permite realizar animaciones de caminar, de apuntar un arma, etc.

## 3.2.5 Menú del juego

Este módulo es el encargado del menú del juego, en el cual se realizará la conexión con el servidor, y se creará una partida nueva o se unirá a una partida ya creada. Este módulo tiene la siguiente organización:




### 3.2.5.1 Clases

La clase Menu es la clase de la cual heredarán todos los menús. Es la encargada de colocar las imágen de fondo y el título del juego.

La clase ServerMenu es la clase encargada de mostrar el menú de conexión con el servidor. Será el primer menú con el que se encontrará el usuario y en este se deberá ingresar el host del servidor y el servicio.

| **ServerMenu** | |
|---|---|
| Atributos: | |
| Gtk::Overlay* menu_container | Contenedor principal del menú |
| Gtk::Image* background | Imagen de fondo |
| unique_ptr<MenuView> next_menu | Puntero al siguiente menú |
| Métodos: | |
| connectToServer(host, service) | Intenta realizar la conexión con el servidor |

La clase MenuView es la clase de la cual heredarán los demás menús. Esta se encarga de mostrar ventanas emergentes al producirse un error fatal o de mostrar un mensaje de error al producirse un error menor.

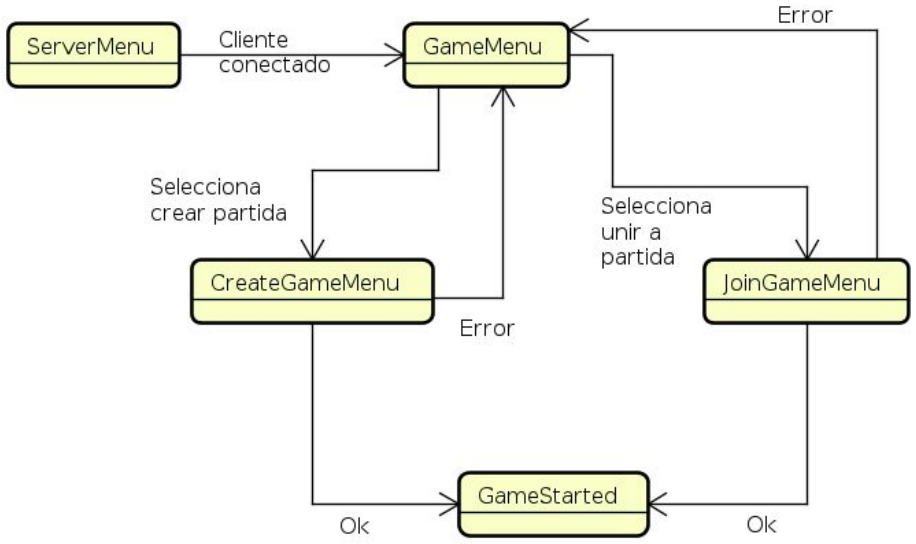
La clase GameMenu se encarga de mostrar el menú de creación del jugador. En este menú el usuario ingresará su nombre y luego podrá ingresar al menú de creación de partida o al menú de unión a una partida.

La clase SelectableListMenu se encarga de agregar en formato de lista las partidas disponibles o los mapas disponibles a una zona específica del menú.

La clase JoinGameMenu se encarga de mostrar todas las partidas disponibles a las cuales se puede unir el jugador.

La clase CreateGameMenu se encarga de mostrar el menú de creación de una partida. En este menú se ingresará el nombre de la partida, la cantidad de jugadores y se mostrarán los mapas que se podrán seleccionar para la partida.

### 3.2.5.2 Diagramas

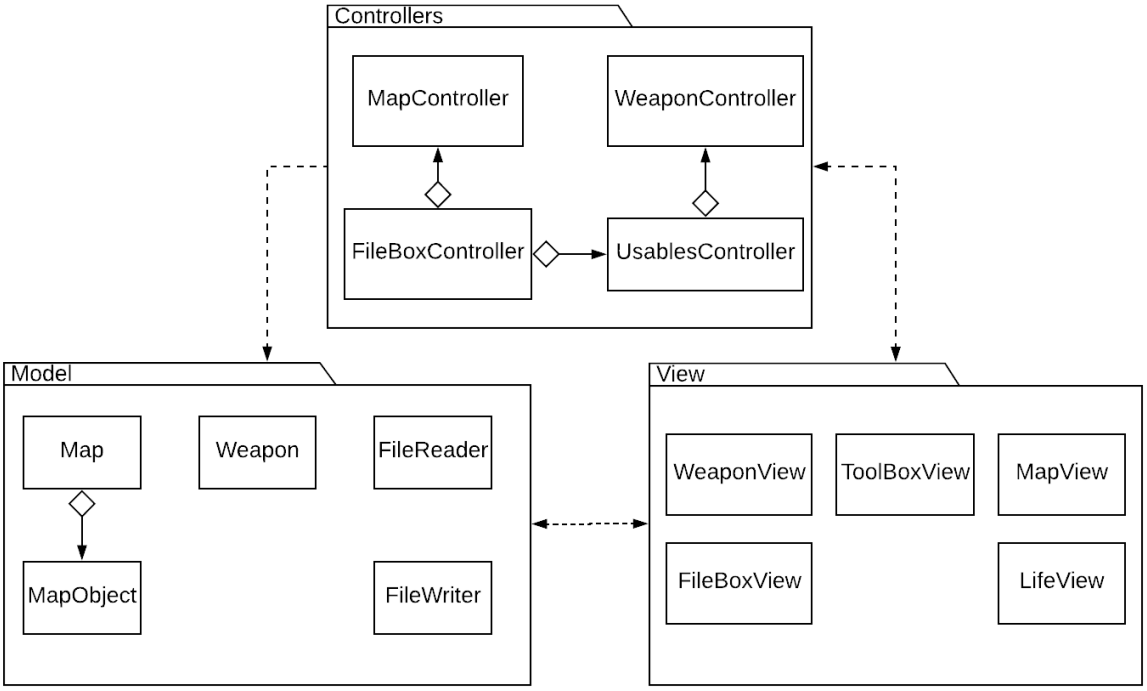



En este diagrama de estados se puede observar como varía el estado de los menús del juego a medida que el usuario va avanzando y seleccionado una acción.

# 3.3 Editor

## 3.3.1 Comunicación interna

El editor está dividido en tres módulos:




Cuando el usuario interactúa con la vista, ésta se lo informa al controlador correspondiente y éste a su vez actualiza el modelo y, si es necesario, también notifica los cambios a la vista.

En las siguientes secciones se explicará con más detalle la interacción entre los distintos módulos.

## 3.3.2 Armas y vida

### 3.3.2.1 Clases

La clase WeaponView está compuesta por los distintos elementos que son vistos por el usuario para cada arma, como lo son el selector de municiones, y una casilla para elegir si se quiere munición infinita, a su vez contiene los estados iniciales de los selectores para cada arma. Cada acción que se realiza sobre esta clase es informada a UsablesController, la cual se detalla más adelante.

| **WeaponView** | |
|---|---|
| Atributos: | |
| Gtk::Scale* ammo_selector | Selector de munición de arma |
| Gtk::CheckBox* infinite | Selector de munición infinita |
| WeaponController* controller | Controlador del arma |
| Métodos: | |
| onAmmoValueChanged() | Ejecutado cuando se cambio el valor de ammo_selector |
| onCheckboxClicked() | Ejecutado cuando se clickea el checkbox infinite |
| resetAmmo() | Resetea a los atributos a su estado inicial |

La clase WeaponController se encarga de actualizar el modelo del arma y, si es necesario, la vista correspondiente según las acciones que haya hecho el usuario.

| **WeaponController** | |
|---|---|
| Atributos: | |
| std::shared_ptr<WeaponView> weapon_view | Vista del arma |
| std::shared_ptr<Weapon> weapon_model | Modelo del arma |
| Métodos: | |
| resetAmmo() | Le informa a la vista y al modelo para que regresen a su estado inicial |
| updateAmmo(new_ammo) | Le informa a la vista y al modelo el nuevo valor que deben tener |

La clase Weapon se encarga de modelar cada una de las armas del juego.

| **Weapon** | |
|---|---|
| Atributos: | |
| const int default_ammo | Valor de munición predeterminado |
| int actual_ammo | Valor actual de munición |
| Métodos: | |
| resetAmmo() | Actualiza el valor del arma a su valor predeterminado |
| setAmmo(new_ammo) | Cambia el valor de la munición actual |

La clase LifeView hereda de Gtk:: SpinButton y contiene el estado inicial de dicho widget, a su vez, informa a UsablesController cuando se produjo una modificación en el mismo.

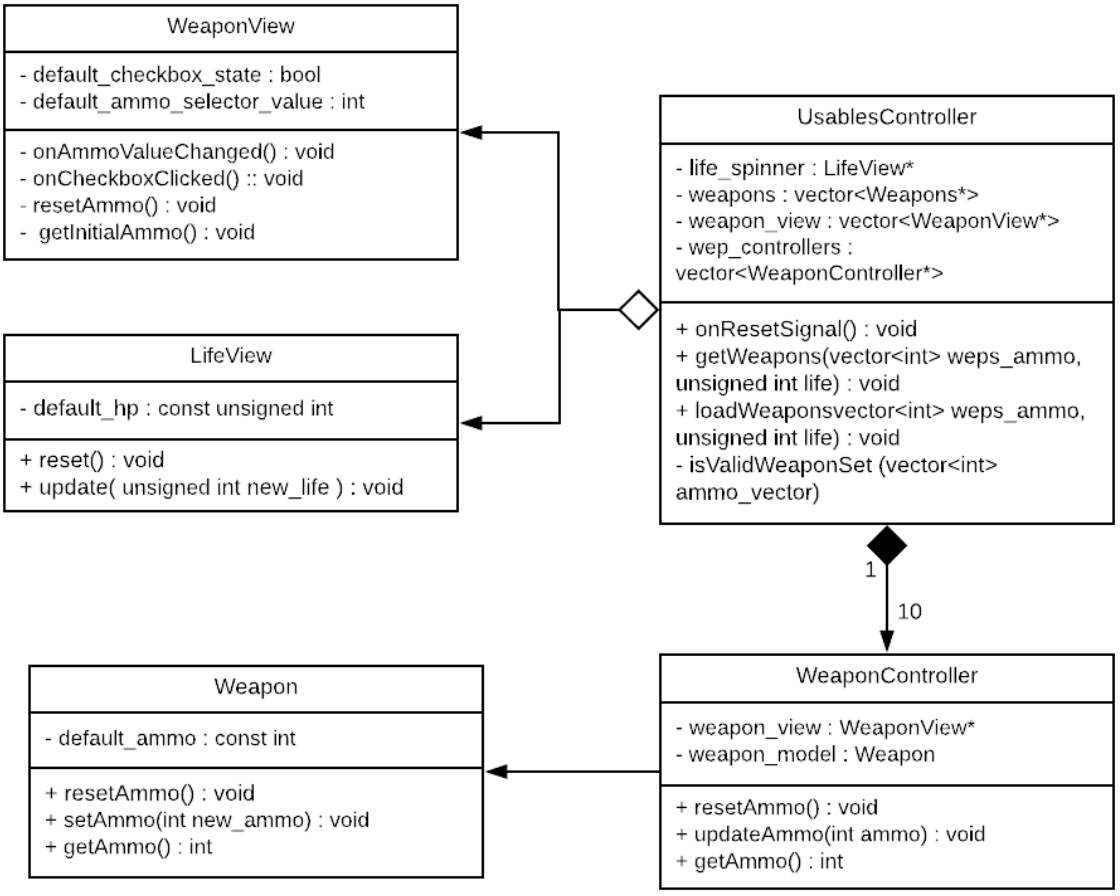
| **LifeView** | |
|---|---|
| Atributos: | |
| const unsigned int default_hp | Valor predeterminado de la vida de cada worm |
| Métodos: | |
| reset() | Vuelve el valor de la vida al valor predeterminado |
| update(new_life) | Setea el valor de la vida al recibido por parámetro |

UsablesController está comunicada con el valor actual de la vida y con las distintas armas. En el caso de un reseteo a valores por defecto, guardado o carga de un mapa, esta clase se encargará de informarle a cada uno de los controladores individuales para obtener o cambiar la información actual que tienen las vistas y los modelos.

| **UsablesController** | |
|---|---|
| Atributos: | |
| LifeView *life_spinner | Selector de la vida de los worms |
| vector<shared_ptr<Weapon>> weapons | Vector que contiene a todos los modelos de las armas |
| vector<shared_ptr<WeaponView>> weapons_view | Vector que contiene a todas las vistas de las armas |
| vector<shared_ptr<WeaponController>> wep_controllers | Vector que contiene a todos los controladores de las armas |
| Métodos: | |
| onResetSignal() | Señal recibida cuando se clickea el boton de reseteo |
| getWeapons(vector weps_ammo,life) | Obtiene todos las municiones de las armas y el valor de la vida |
| loadWeapons(vector weps_ammo,life) | Sete todos las municiones de las armas y el valor de las armas |

### 3.3.2.2 Diagramas

En el siguiente diagrama se pone en evidencia la relación entre las clases mencionadas anteriormente.



## 3.3.3 Mapa

### 3.3.3.1 Clases

La clase Map tiene como objetivo modelar el mapa que está siendo editado, hereda de Gtk::Layout

| **Map** | |
|---|---|
| Atributos: | |
| vector<pair<int, MapObject>> contained_objects | Vector que contiene los objetos agregados al mapa |
| Métodos: | |
| clean() | Limpia todos los objetos que se encontraban actualmente en el mapa |
| erase(new_life) | Setea el valor de la vida al recibido por parámetro |
| add(id, x, y, angle) | Agrega un objeto nuevo del tipo id al mapa |
| move(index, x, y) | Mueve el objeto en la posición index del vector hacia la posición (x,y) |
| turn(id, index, angle) | Gira el objeto en la posición index del vector en el ángulo indicado |

La clase MapView se encarga de manejar la vista del mapa actual

| **MapView** | |
|---|---|
| Atributos: | |
| vector<Gtk::Image> contained_objects | Vector que contiene las imágenes que se fueron colocando en el mapa |
| vector<Gtk::Image> background | Vector que contiene las imágenes de fondo |
| ScrollHandler scroll_handler | Clase que se encarga de manejar los desplazamientos al acercarse a los bordes del mapa |
| Métodos: | |
| onButtonClicked() | Se llama a este método cuando se clickea sobre el mapa |
| select(x, y) | Selecciona la imagen que se encuentra en la posición (x,y), si es que la hay |
| changeBackground() | Cambia el fondo actual que tiene el mapa |

| | |
|---|---|
| clean() | Borran todos los objetos que tiene el mapa |

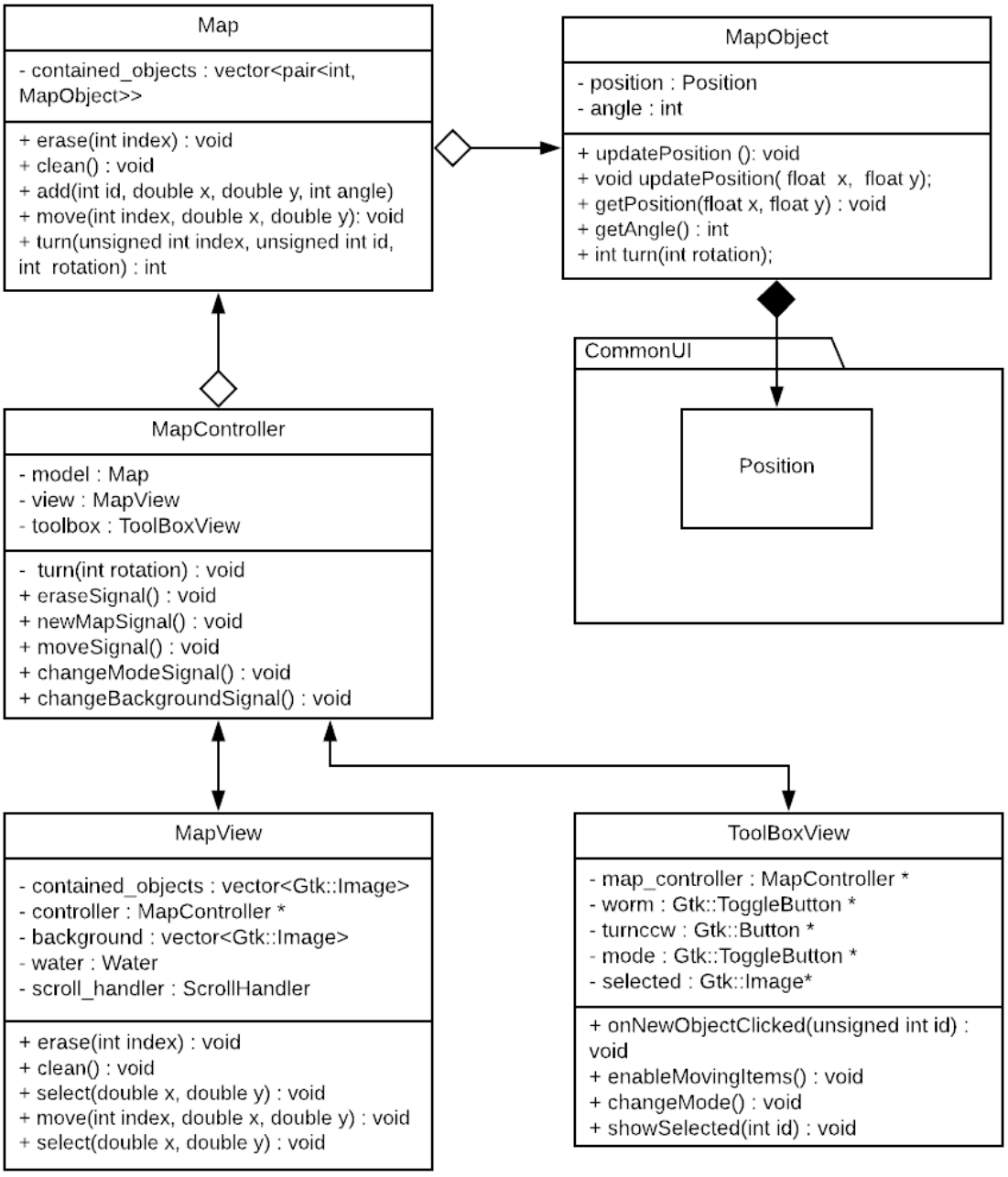
MapController está compuesta por el modelo y la vista del mapa, también está en comunicación con la clase ToolBoxView para recibir o informar cualquier cambio que se haga en la vista de la botonera.

| **MapController** | |
|---|---|
| Atributos: | |
| Map model | Modelo del mapa |
| MapView *view | Vista del mapa |
| ToolBoxView *toolBox | Puntero a la botonera que se encuentra en la parte izquierda del mapa |
| Métodos: | |
| addModeSignal() | Ejecuta este método cuando se elige algún objeto para agregar |
| changeModeSignal() | Se ejecuta al clickear el botón de selección |
| newMapSignal() | Se ejecuta al clickear el botón de nuevo mapa |
| moveSignal() | Se ejecuta cuando se clickea el botón de mover y se encuentra en el modo selección |
| eraseSignal() | Se ejecuta cuando se clickea el botón de borrar y se encuentra en el modo selección |

La clase toolBoxView hereda de Gtk::Grid y está compuesta por todos los widgets que componen la botonera de la parte izquierda, es decir, los botones que permiten realizar acciones al usuario. Se comunica con MapController cuando se selecciona alguno de sus elementos.

### 3.3.3.2 Diagramas

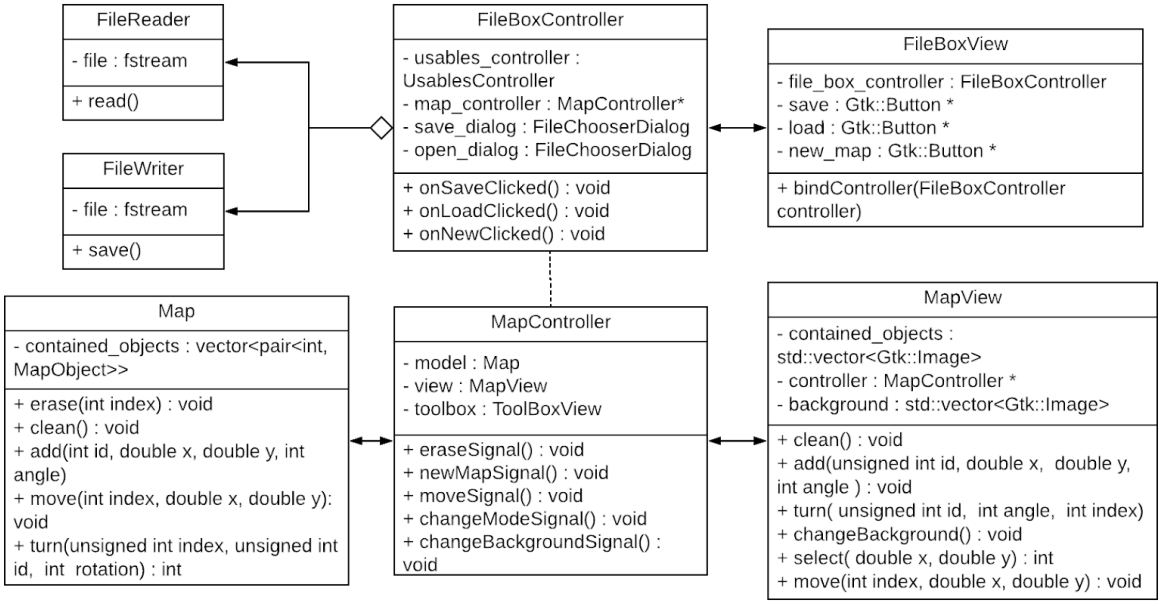
En el siguiente diagrama se muestra la relación entre las clases nombradas anteriormente.



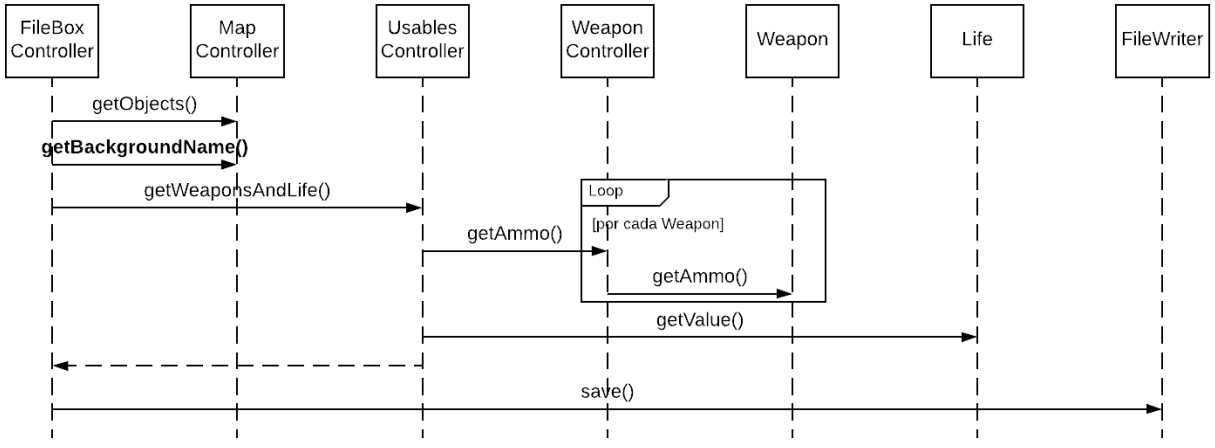
## 3.3.4 Guardado y carga del mapa

Cuando el usuario comienza el proceso de guardado, en primer lugar se valida el mapa actual, en caso de no ser válido se lanza una excepción InvalidMapError que recibe como argumento el mensaje a ser mostrado en el cuadro de diálogo que genera dicha excepción.

En caso de que el mapa sea válido, la clase FileBoxController inicia el proceso de guardado, en primer lugar muestra un diálogo de selección de archivos en el cual se indicará cual es la ubicación que tendrá el nuevo mapa guardado.




Una vez seleccionada la ubicación, FileBoxController enviará toda la información recibida desde MapController y, de forma similar, desde UsablesController, hacia la clase FileWriter, como se puede ver en el próximo diagrama.



La clase FileWriter se encarga de pasar toda la información que se ha recolectado del mapa actual hacia un archivo en formato Yaml.

De forma similar al guardado, cuando el usuario desee cargar un mapa, la clase FileBoxController mostrará un diálogo de selección de archivos en el cual el usuario deberá elegir el mapa a cargar.

Luego de dicha selección, el FileBoxController le pasa el nombre de archivo a la clase FileReader, la cual se encarga de extraer toda la información acerca del mapa y pasarla a FileBoxController. La información acerca de los objetos colocados en el mapa se deriva al MapController y la que respecta a la vida y las armas, hacia UsablesController. Cada controlador a su vez la derivará a cada modelo y vista correspondiente.