Taller de Programación I (75.42)
Facultad de Ingeniería
Universidad de Buenos Aires

# Wolfenstein 3D

## Ejercicio Final

## Documentación Técnica

## 2º Cuatrimestre 2020

**Primera Entrega: 09/02/2021**

**Segunda Entrega: 23/02/2021**



| Alumnos | Padrón |
|---|---|
| Carretero Andres | 101004 |
| Montiel Ramiro | 100870 |
| Parente Gaston | 101516 |
| Roldan Maria Cecilia | 101939 |

# Índice

# Requerimientos de Software

- Se requiere utilizar un sistema operativo Linux.
- Las siguientes dependencias deben ser instaladas :
  - coreutils
  - cmake
  - build-essential
  - qt5-default
  - libsdl2-dev
  - libsdl2-image-dev
  - libsdl2-ttf-dev
  - liblua5.3-dev
  - libyaml-cpp-dev
  - libsdl2-mixer-dev

# Descripción general

La aplicación consta de 3 módulos principales: Servidor, Cliente y Editor. Pero se detallaran 4 módulos ya que también creamos un módulo Común tanto para Servidor como para Cliente.

# Módulo Cliente

## Descripción general

La totalidad de la lógica del juego así como del almacenamiento de todos los elementos del mismo se encuentran en la clase GameModelClient, el cual crea los posicionables y el Screen al inicio del juego y procesa los protocolos para actualizarlos.
El loop del juego se encuentra en UserClient, donde cada 30 milésimas de segundo recibe las teclas presionadas por el usuario, las envía por medio de protocolos al servidor, recibe otros protocolos del servidor y los envía a ser procesados, para luego mostrar la pantalla actualizada.

Los principales objetos del GameModelClient son el Screen,  Textures, Player, Enemies, y el resto de objetos del mapa, como Paredes, Puertas y Sprites

## Clases

- ClentMap:
  Es el mapa del juego, almacena punteros a los posicionables almacenados en cada posición y se encarga de mover los posicionables Movibles cuando sea necesario.

- Texture: almacena todas las texturas que se utilizarán en el juego, tanto de paredes, enemigos, armas, sprites, o la barra de información. Además esta es la clase a la que llaman todos los Posicionables para graficarlos.

- SpriteDrawer: contiene todos los elementos necesarios y las funciones para dibujar los sprites.

- Screen: Durante el loop del juego es llamada una vez por frame, en este se ejecuta el Raycasting, donde cada rayo sigue hasta colisionar con una pared, o puerta, y grafica la textura de la posición exacta con la que chocó. Previo a esto pudo haber chocado con otros elementos (sprites, enemigos, o misiles), en cuyo caso se los marca como “avistados” para que al finalizar el Raycasting se realice un procedimiento con éstos elementos para graficarlos de igual manera.

- Raycasting: crea todos los rayos que se van a necesitar en este frame.

- Ray: es inicialmente creado con el angulo del rayo, su posición inicial(la del jugador), el mapa para obtener los objetos y el numero del rayo para saber qué pixel de la pantalla hay que dibujar, cada vez que choca con un nuevo casillero le solicita al mapa el puntero al Posicionable de esta posición, de ser nulo (porque está vacía), continúa, en caso contrario llama al método colisioned de este Posicionable, que según el tipo que sea, ‘retorna’ el método doorColided, donde dibujo la puerta,

wallColided, donde dibujo la pared, o spriteColided, donde marco al sprite como avistado en este frame y continúo buscando.

- Posicionables: Todos los objetos que se almacenan en el mapa son hijos de esta clase abstracta, posee sus coordenadas y un puntero a Texture.

Sus clases hijas son:

- Wall: posicionable abstracto, todos los distintos tipos de paredes heredan de éste.

- SpriteHolder: posicionable que almacena los sprites que se encuentran en esa posición en el mapa, pueden ser recolectables (armas, botiquines, llaves), o de entorno (lámparas, barriles); colisionables (pilares, mesas) o atravesables (charco de sangre o las mismas armas si el jugador ya las tiene). La distinción de colisionables o atravesables es necesaria porque esta clase, ademas de almacenar múltiples sprites, en caso de que el SpriteHolder sea atravesable, será también capaz de almacenar objetos movibles (el jugador, los enemigos, o los misiles), por lo que cuando uno de estos se mueva en el mapa de una posicion vacia, a una posición ocupada por un SpriteHolder en el cual puedan meterse, este movible dejará de tener una posición propia en el mapa, y el SpriteHolder en el que se encuentra ahora tendrá un puntero a éste hasta que se mueva a otro casillero. Por lo tanto, cuando ese SpriteHolder sea avistado durante el raycasting, tambien le indicara al movible que está almacenando que tambien lo fue. Además, y dado que las explosiones tambien son basicamente sprites, ésta clase tambien las muestra: cuando un misil explota, de encontrarse justo sobre un sprite, setea a true que está almacenando una explosion y muestra su animacion hasta que finaliza, en caso contrario de que el misil explote en un casillero vacío, se crea un sprite en ese mismo casillero para mostrar la explosion y una vez termina, se elimina el objeto

- Door: posicionable de comportamiento muy similar a las Paredes, pero que estando abierta, tambien puede almacenar un objeto Movible al igual que SpriteHolder.

- Movable: posicionable abstracto, almacena el mapa para que los objetos de este tipo muevan cambiar su posicion en el mapa internamente.

- Rocket: posicionable de tipo movible y tambien es un SpriteDrawer, seteado con una posición inicial y se va modificando a medida que avanza.

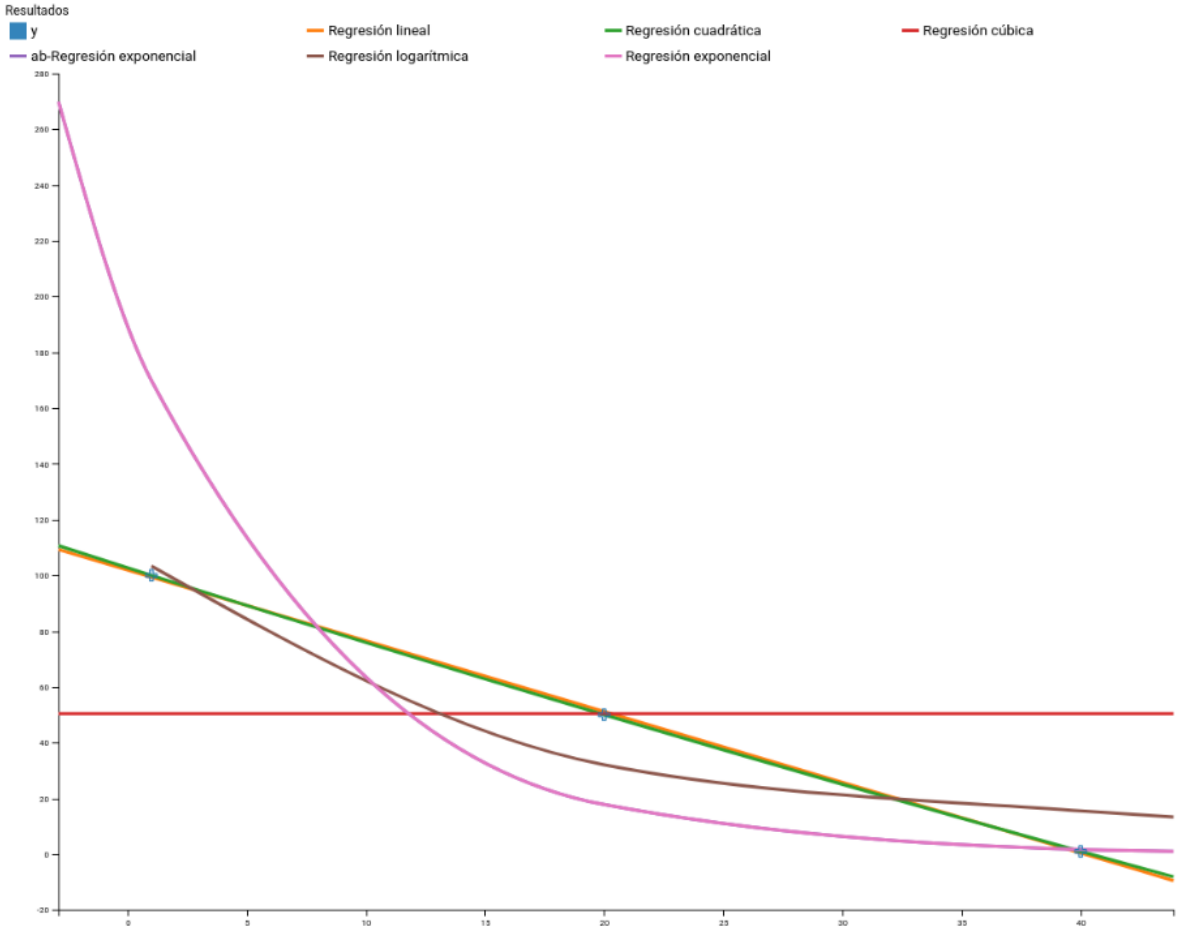
- Character: posicionable abstracto de tipo movible, como los únicos posicionables de este tipo pueden ser Enemy y Player, almacena la dirección a la que estos estan mirando y sus posiciones y direcciones nuevas para que revivan siempre en el mismo lugar.

- Player: el jugador que controla el cliente, es un Character, almacena todos los datos del cliente y tiene un arma.

- Enemy: posicionable de tipo Character que, dado que se dibuja de la misma manera que los sprites, tambien hereda de SpriteDrawer.

- Sound: Clase que almacena el procesamiento de sonidos. Utiliza SDL para hacer sonar el sonido que se le indica en el constructor mediante el path de dicho archivo de sonido. Contiene un std::unique pointer para simplificar la destrucción del mismo.

- SoundPlayer: Clase que wrappea el uso de la Clase Sound y almacena todos los sonidos de animaciones utilizados durante el juego. Como puntos importantes encontramos en esta clase el método que permite modular el volumen del sonido en función de la distancia, utilizando una función exponencial para calcularlo:




De todas las pruebas que hicimos, la que mejor se adaptó a la sensación de distancia al oído fue la regresión exponencial.

- BackgroundMusic: Clase encargada de la música de fondo durante el juego, el funcionamiento de la misma consiste en reproducir aleatoriamente los archivos .mp3 o .wav que se encuentren en la carpeta data/background_music/ si por algún motivo queremos mutear la música, simplemente debemos presionar la tecla M durante la partida para activarla o desactivarla, esto funciona gracias al método togglePause().

Clases de estados:

- **GunType**: clase abstracta. El jugador tiene un puntero a un objeto de este tipo, que a su vez es un objeto de alguno de los tipos de armas disponibles, los cuales varían segun el arma que esté usando, lo que influye en la cantidad de frames que muestro la animacion de disparo, si puedo disparar de forma contínua, y el método para dibujar que llamo de texture.
  Tipos de armas:
    Knife
    Gun
    MachineGun
    MissileLauncher

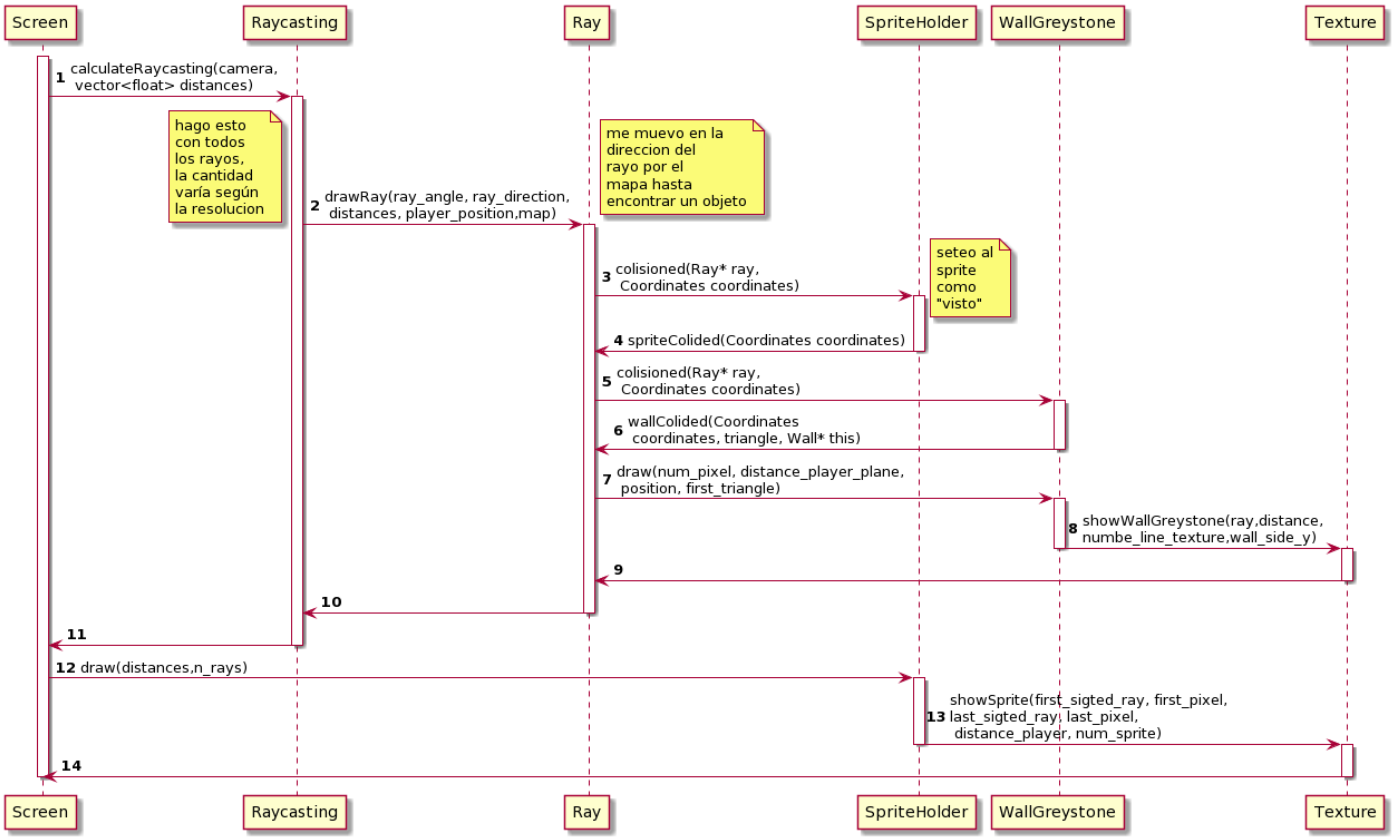
- **EnemyType**: clase abstracta. Cada enemigo tiene un puntero a un objeto de este tipo, que a su vez es un objeto de alguno de los tipos de enemigos existentes, los cuales varían segun el arma que esté usando, lo que influye en la cantidad de frames que muestro la animacion de movimiento, y el metodo para dibujar que llamo de texture.
  Tipos de enemigos:
    Dog
    Guard
    Ss
    Officer
    Mutant

# Muerte

Para el caso de los jugadores que agotan todas sus vidas pero que aún quede tiempo para el fin de la partida porque quedan dos o más jugadores vivos, se implementó el modo espectador, o modo fantasma. Consiste en, a grandes rasgos, que cuando el jugador termina todas sus vidas, se setea la variable en el GameModelClient de player_alive a false, y ante esto en el loop del juego de UserClient, el jugador dejará de enviar sus protocolos al servidor (pues ya está muerto y nada de lo que haga afecta al resto), se lo removerá del mapa, y luego se comenzará a usar un método de movimiento distinto: en el cual el jugador, podrá moverse libremente por el mismo.
Este tipo de movimiento implica modificar la posicion y direccion interna del jugador, pero sin agregarlo en ningún momento al mapa de juego, por lo que podrá atravesar paredes, objetos e incluso enemigos dado que nunca colisionará con nada y que para las técnicas del raycasting y dibujado de sprites se usa directamente la posición y dirección interna del jugador aún si no está posicionado en ningún casillero del mapa, y poder ver cómo de desarrolla el fin de la partida sin afectar al resto.

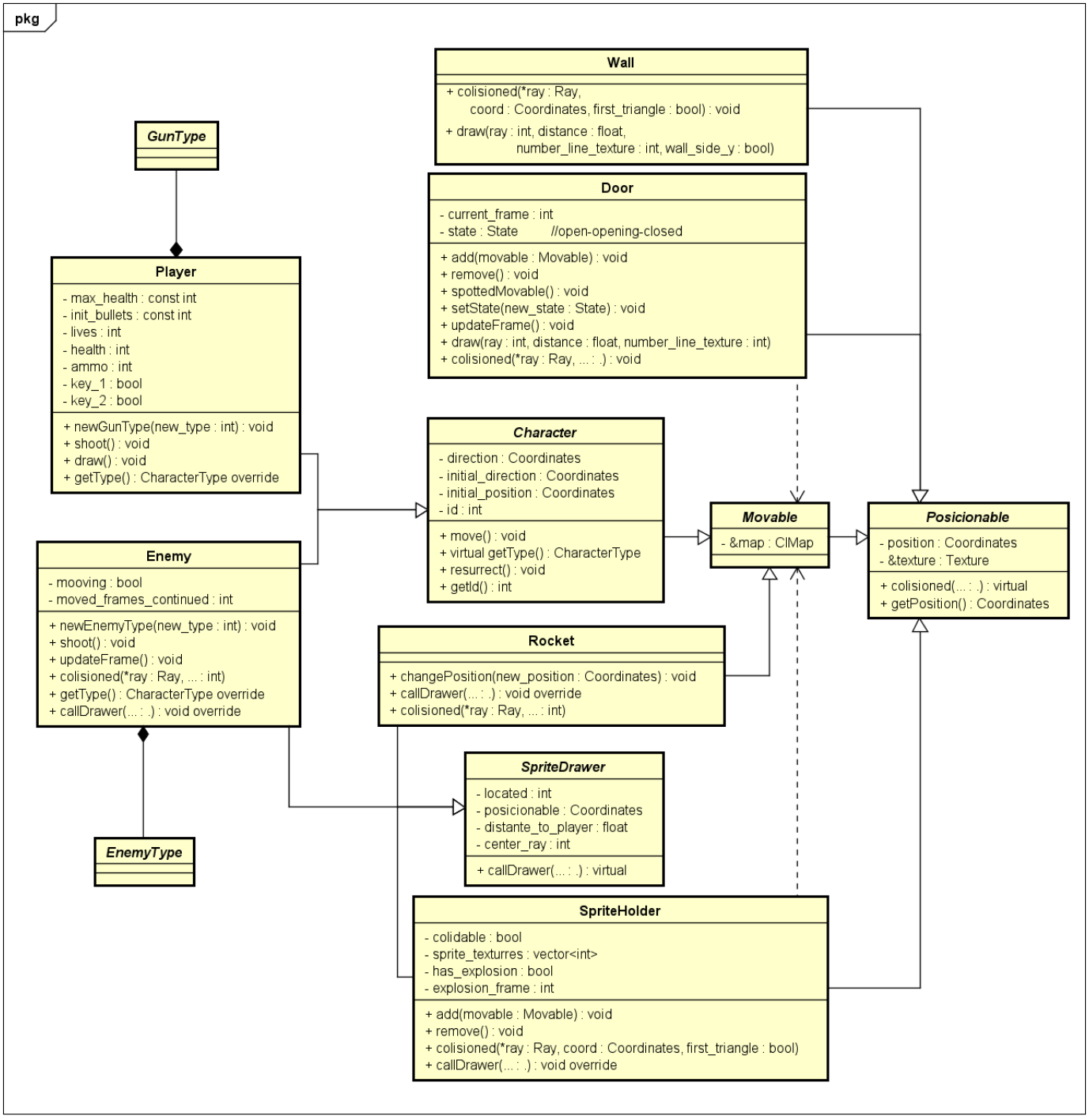
# Diagramas

En este diagrama se puede ver el funcionamiento básico de la interfaz,en cada frame Screen le envia al Raycasting la cámara y un vector vacío de distancias, en este caso a modo de ejemplo se emite un rayo que inicialmente choca con un sprite, setea al sprite como “avistado” y sigue hasta encontrar una pared (o puerta). Una vez hecho esto, se agrega al vector de distancias la distancia entre la posición exacta de la pared donde chocó el rayo con el plano del jugador, y se grafica ésta franja de pixeles en la pantalla. Se repite el procedimiento una vez por cada rayo, es decir, por pixel de ancho de resolución.
Terminado esto y devuelta en Screen, se grafica, por orden de distancia al jugador, los sprites avistados en este frame, se determina qué rayos chocarían contra el mismo, y se grafican aquellos rayos cuya distancia del sprite al jugador, no superen la distancia de ese mismo rayo a la pared mas cercana, almacenada justamente en el vector de distancias.

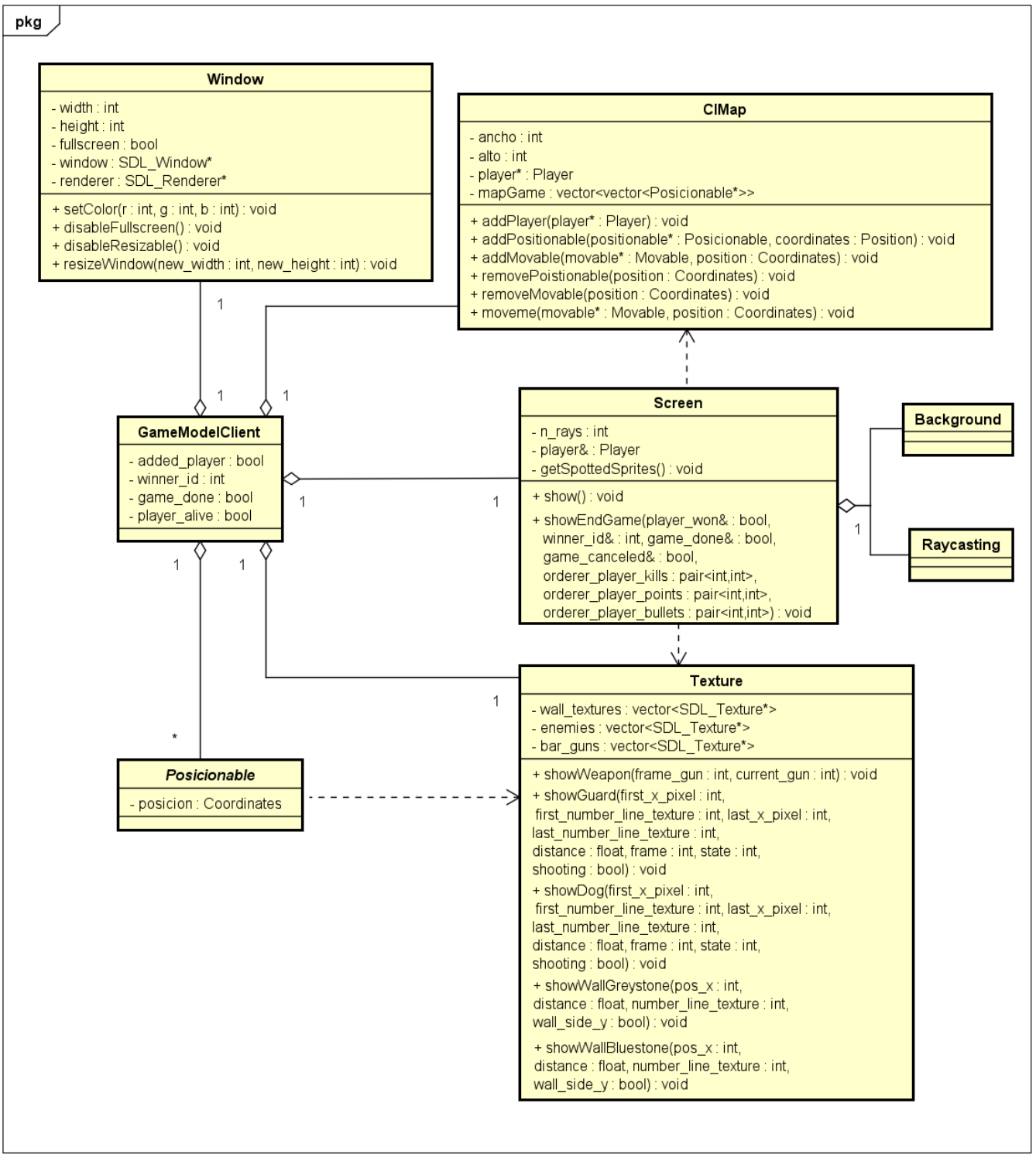



En este diagrama se aprecia con mayor claridad los distintos tipos de posicionables existentes, y la diferencia entre los abstractos y los concretos. Por lo que en realidad los únicos tipos de Posicionables que verdaderamente se puede tener son Wall, Door, Rocket, Player, Enemy y

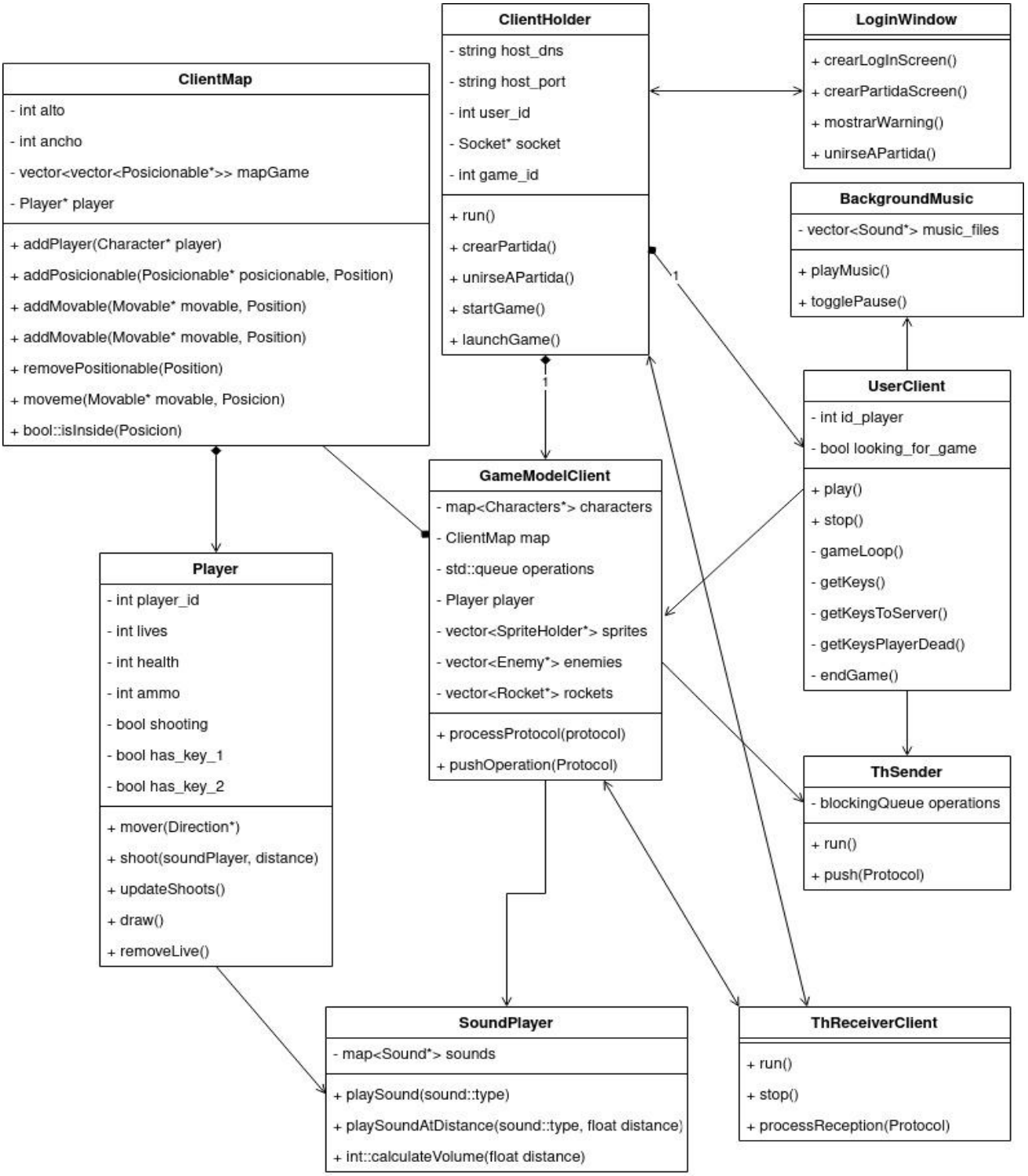
SpriteHolder



Acá se ve que en el GameModelClient se maneja la totalidad de las partes graficas del juego, el mismo almacena todos los Posicionables, crea el Windows, el mapa, el Screen y Texture.



ClientHolder: El client holder al contener a todo el programa, es el que se encarga de ir creando a todas las clases que utilizamos en el cliente a medida que el juego se va desarrollando, inicialmente crea la clase LoginWindow, luego, el GameModelClient y el UserClient y asi sucesivamente. Cabe destacar que toda la biblioteca SDL se ejecuta en el Thread principal, mientras que los threads ThReceiverClient y ThSender son lanzados por éste ClienteHolder



# Descripción de archivos y protocolos

## Archivos

- En el programa cliente, se utilizan todos los archivos que se encuentran en la carpeta /data/textures. Los cuales son utilizados para dibujar todos los sprites del juego, desde los tipos de enemigos, hasta las paredes, puertas, items, etc.

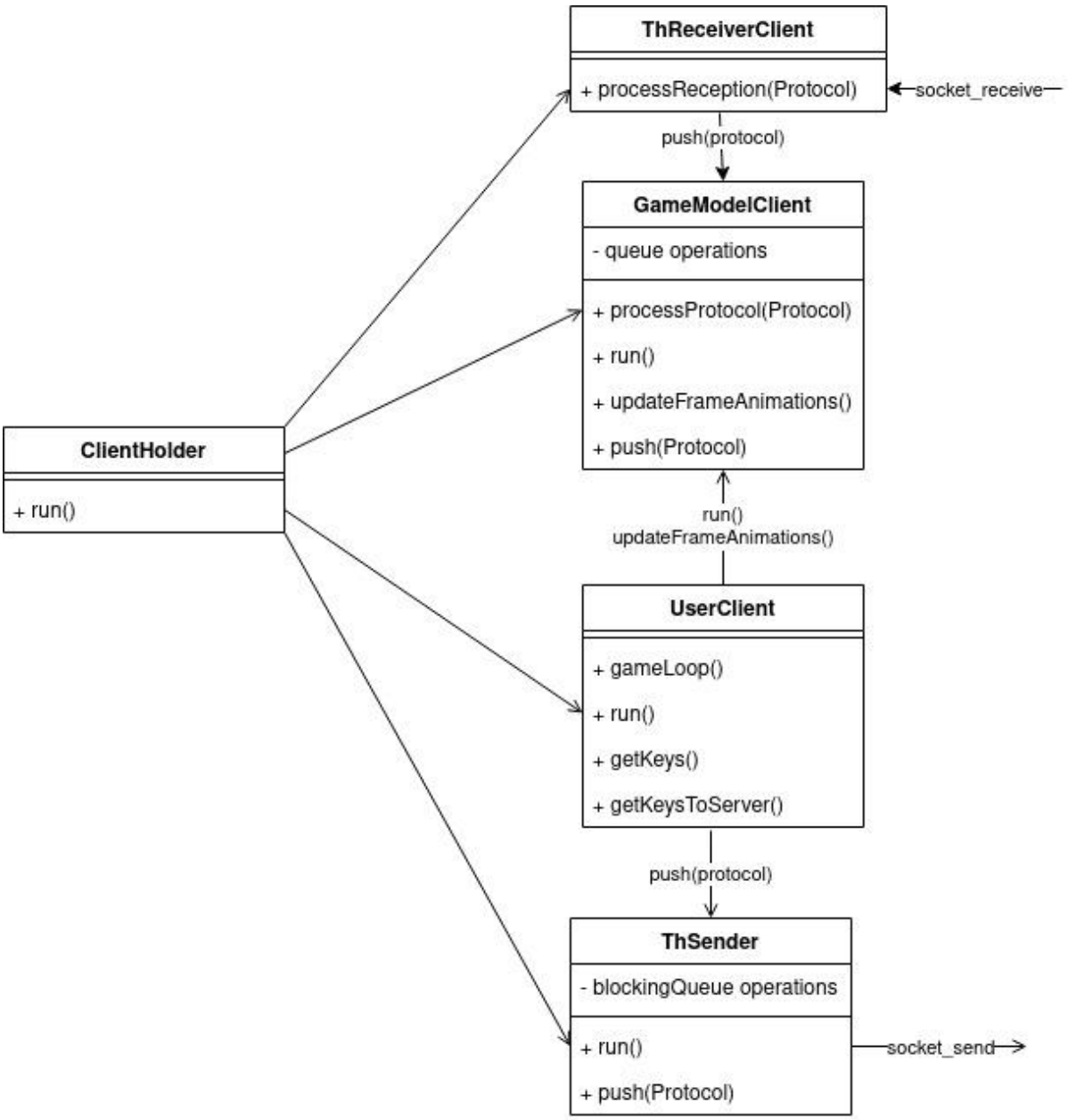
Por ejemplo encontraremos las siguientes imágenes:

- Barriles 
- Paredes 
- Municiones 
- Puertas sin llave 

- A su vez en la carpeta /data encontraremos la imagen de fondo de la pantalla de inicio.
- En la carpeta /data/sounds se encuentran todos los efectos de sonido, como ruidos de armas, puertas, etc.
- En la carpeta /data/background_music encontraremos los tracks que se ejecutan de fondo cuando jugamos una partida.
- En la carpeta /data/maps encontraremos los mapas que tenemos en nuestra computadora.

## Protocolos

Desde el programa cliente se envían y reciben protocolos por red utilizando sockets TCP
A continuación veremos  la comunicación internamente en el cliente y hacia el servidor:




Como vemos, el programa cliente recibe Protocolos desde el servidor y mediante el ThReceiverClient, son insertados en la cola de operaciones del GameModelClient, mientras tanto, el UserClient dibuja utilizando SDL, al momento de dibujar lo que hace es decirle al GameModelClient que actualice todas sus operaciones y actualice los frames de cada sprite para luego dibujar, luego se evalúa si se presionaron teclas de teclado, en caso de que esto ocurra, se crea un protocolo con la correspondiente acción y se la agrega mediante un push a la cola de operaciones del ThSender, el cual enviará al servidor dicho Protocolo.

# Módulo Común

## Descripción general

Al momento de realizar el trabajo, nos dimos cuenta que había muchas clases que se iban a requerir tanto el en programa cliente como en el programa servidor, por eso podemos encontrar un apartado con todas las clases que se utilizan en ambos programas.

## Clases

- BloquingQueue:
  Clase encargada de administrar y representar una cola bloqueante que luego será utilizada por los GameModels tanto del cliente como del servidor.

- ConfigVariable:
  Archivo que guarda la variable global Configs. Esta variable es cargada en función de la información leída del archivo de configuración en el servidor y apenas un usuario se conecta, es transmitida por red al mismo.

- Coordinates:
  Clase que guarda la lógica de coordenadas. Almacena las posiciones (x,y), las direcciones en las que miran los jugadores y maneja la lógica aritmética al momento de desplazarse y al momento de calcular ángulos.

- Direction:
  Clase utilizada para realizar double dispatch al momento de moverse por parte de los jugadores o rockets. Funciona de forma polimórfica para que el jugador simplemente diga que quiere moverse hacia alguna dirección, cada una de las direcciones: DirForward para ir hacia adelante, DirBackwards para ir hacia atrás, etc. sabe que tiene que hacer para retornar las coordenadas correctas de la posición a la que el jugador debe moverse.

- GameModel:
  Clase principal de los programas cliente y servidor, ella guarda un mapa de todas las direcciones para poder acceder en un tiempo O(1) a cada una de estas y también almacena el ID propio de la partida que se va a jugar.

- MapLoader:
  Clase implementada para poder corroborar que el cliente y el servidor están utilizando el mismo mapa. Si se la crea pasándole un string de nombre de mapa realiza un hash MD5 (recortado a sus últimos 16 bits) de dicho nombre. Si se la crea pasándole un hash MD5 (recortado a sus últimos 16 bits), buscará en la carpeta de mapas si alguno cumple con dicho hash. De esta forma resolvimos el problema de consistencia entre el mapa que lee el servidor y el que lee el cliente, sean el mismo.

- Posicionable:

Clase padre encargada de guardar a todos los objetos que se van a encontrar en el mapa, almacena la posición y dirección de cada uno.

- Protocol:
  Clase encargada de manejar todas las comunicaciones por red entre el cliente y el servidor, al momento de utilizarlo dentro del cliente y/o el servidor, se utiliza esta clase, pero cuando se la envía por red, se genera un struct llamado NetworkProtocol, el cual tiene los atributos pasados por las funciones htons ó htols y cuando se lo recibe se crea un nuevo protocolo utilizando los valores del NetworkProtocol recibido pero pasando la función ntohs ó ntohl a los atributos de este.

- Socket:
  Clase que almacena el atributo file descriptor correspondiente al socket de comunicación entre cliente y servidor, también guarda todos los métodos para conseguir crear un socket y bindearlo a un servidor si se encuentra desde un programa cliente y para escuchar y aceptar clientes si nos encontramos desde un programa servidor.
  Heredan de el SocketClient y SocketServer para una clara utilización de los mismos.

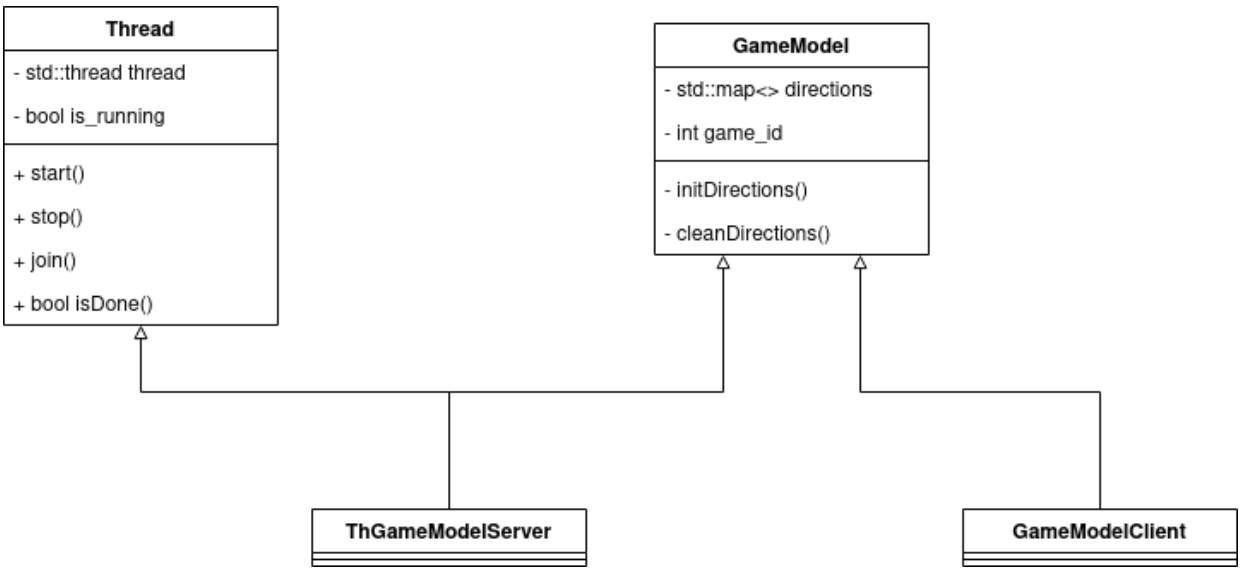
- Thread:
  Clase padre encargada de encapsular el comportamiento de otras clases (que heredarán de esta) para poder ser ejecutadas en un hilo independiente.

- ThreadReceiver:
  Clase padre que hereda de Thread y que contiene un socket, la cual se encarga de recibir información y enviarla al objeto correspondiente de procesarla ya sea gameModel o User de forma concurrente.

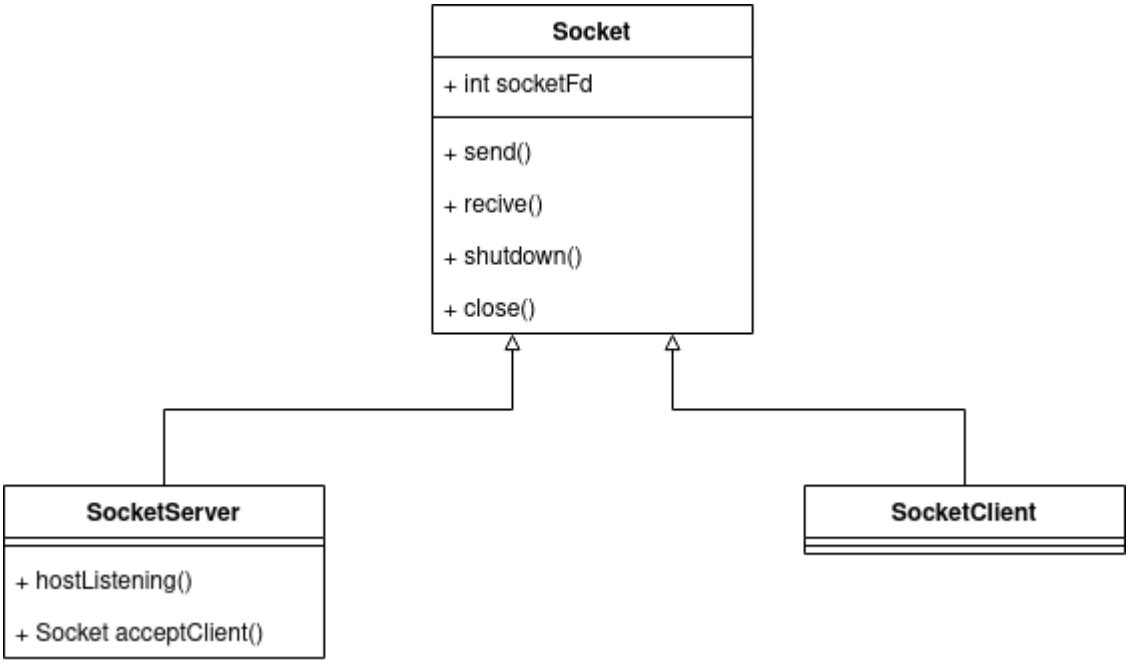
- ThreadSender:
  Clase padre que hereda de Thread y que contiene un socket, la cual se encarga de enviar la información que encuentra en su cola de operaciones a transmitir.

# Diagramas UML

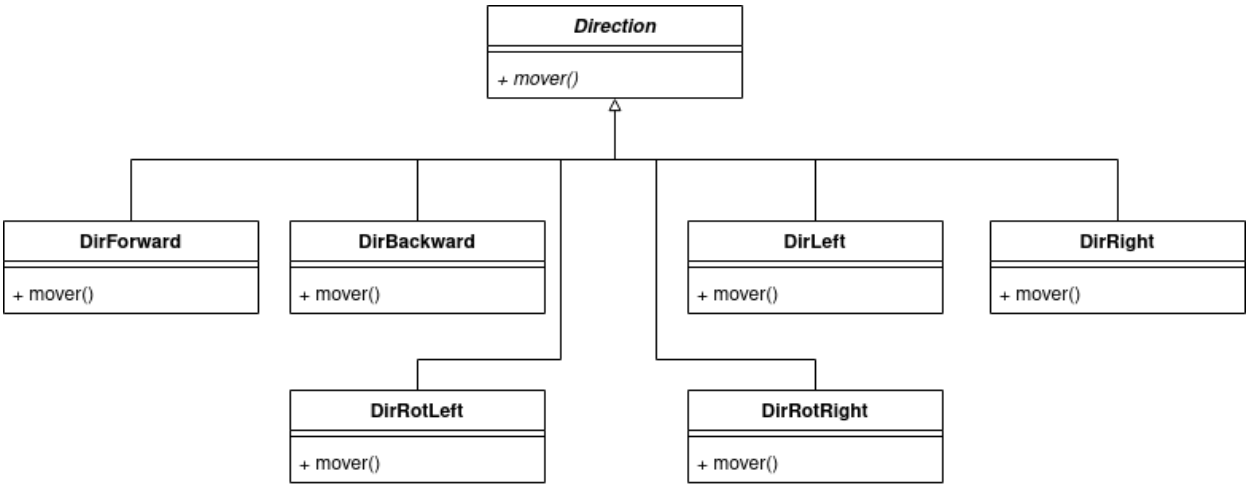
Thread y Game Model



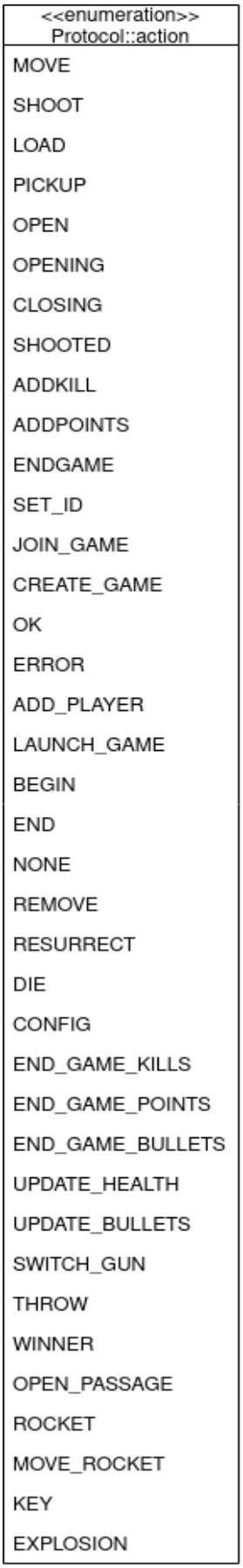

Socket:

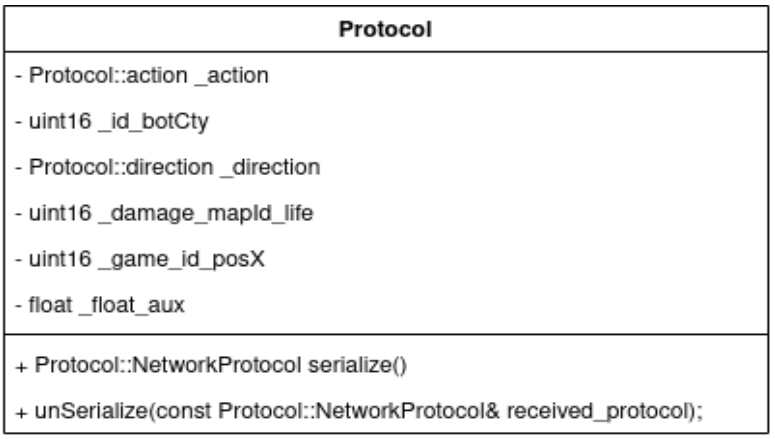


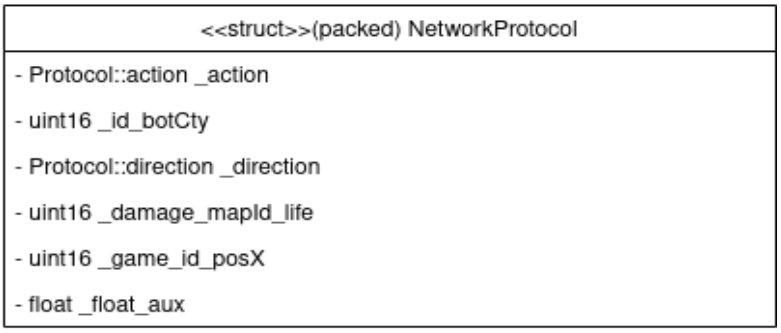

Direction:

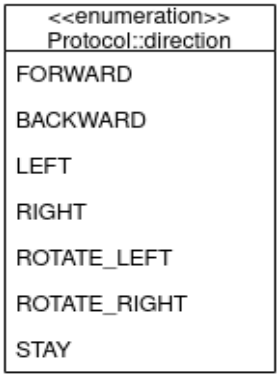


Protocol:


<<enumeration>>
Protocol::action
MOVE
SHOOT
LOAD
PICKUP
OPEN
OPENING
CLOSING
SHOOTED
ADDKILL
ADDPOINTS
ENDGAME
SET_ID
JOIN_GAME
CREATE_GAME
OK
ERROR
ADD_PLAYER
LAUNCH_GAME
BEGIN
END
NONE
REMOVE
RESURRECT
DIE
CONFIG
END_GAME_KILLS
END_GAME_POINTS
END_GAME_BULLETS
UPDATE_HEALTH
UPDATE_BULLETS
SWITCH_GUN
THROW
WINNER
OPEN_PASSAGE
ROCKET
MOVE_ROCKET
KEY
EXPLOSION


Protocol
- Protocol::action _action
- uint16 _id_botCty
- Protocol::direction _direction
- uint16 _damage_mapId_life
- uint16 _game_id_posX
- float _float_aux
+ Protocol::NetworkProtocol serialize()
+ unSerialize(const Protocol::NetworkProtocol& received_protocol);


<<struct>>(packed) NetworkProtocol
- Protocol::action _action
- uint16 _id_botCty
- Protocol::direction _direction
- uint16 _damage_mapId_life
- uint16 _game_id_posX
- float _float_aux


<<enumeration>>
Protocol::direction
FORWARD
BACKWARD
LEFT
RIGHT
ROTATE_LEFT
ROTATE_RIGHT
STAY

# Módulo Servidor

## Descripción general

El programa servidor es el que se encarga de toda la lógica correspondiente al juego. Procesa todos los protocolos con peticiones por parte de los clientes, los procesa y le responde a cada jugador la acción que debe realizar según el resultado que obtuvo. También es capaz de soportar múltiples partidas con múltiples jugadores a la vez y opcionalmente soporta agregar a cada partida una cantidad de bots.

## Clases

- ServerHolder:
  Clase encargada de contener a todo el programa servidor.

- GamesAdmin:
  Clase encargada de manejar la creación y destrucción de partidas, también maneja el ingreso de jugadores a cada partida.

- ParamReaderServer:
  Clase encargada de verificar los parámetros ingresados al lanzar la aplicación servidor y en caso de ser correctos, parsea el archivo de configuración con extensión YAML y carga la variable global configs.

- ServerMap:
  Clase encargada de almacenar las posiciones de todos los jugadores que se encuentren en una partida.

- ThAcceptor:
  Clase que hereda de Thread cuya función es aceptar nuevas conexiones al servidor e inmediatamente enviarles los parámetros de configuración leídos del archivo de configuración a cada nuevo usuario.

- ThBots:
  Clase que hereda de Thread que se encarga de controlar y regular el tiempo de uso de CPU por parte de los bots que se encuentren en la partida.

- ThGameEvents:
  Clase encargada de procesar y enviar protocolos al ThGameModelServer para que avise al resto de los clientes los eventos: Apertura, cierre de puertas y tiempo de finalización de juego.

- ThGameModelServer:
  Clase principal al momento de jugar, esta clase se encarga de procesar todos los protocolos enviados por todos los jugadores durante una partida, luego de procesar un protocolo recibido, envía a los jugadores correspondientes un protocolo de respuesta para que se actualice en sus modelos la información correspondiente.

- ThReceiverServer:
  Clase que hereda de ThReceiver, con métodos especializados para enviar los protocolos recibidos por red tanto al ThGameModelServer como al ThUserServer.

- ThUserServer:
  Clase que representa a cada jugador del lado del servidor, es mayormente utilizada durante los momentos previos al juego, esta clase procesa todos los protocolos correspondientes a la creación y unión de jugadores a la partida corriente.

- Player:
  Clase encargada de representar a un jugador del juego. Guarda información de su estado, como su vida, balas, llaves, posición en el mapa, estado de soldado, e información del cliente al cual representa, id del jugador. Se encarga de realizar las acciones determinadas por el cliente, puede moverse, disparar, cambiar de arma, levantar ítems del mapa, entre otras. También realiza acciones específicas del juego, como morir y revivir.

- SoldierState:
  Guarda el soldado que contiene el arma actual del jugador. Controla sus armas, cambiando de soldado de ser necesario, actualizando las balas y agregando nuevos soldados en caso de levantar armas del mapa.

- Soldier:
  Es la clase de la cual heredan los distintos soldados. Se encarga principalmente de disparar, verificando los ángulos entre los jugadores y sus distancias, y causando daño en los jugadores atacados. Las clases heredadas de Soldier son:
    - Dog: ataca con un cuchillo a corta distancia.
    - Guard, SS, Officer: disparan balas en su rango de disparo, este rango depende de su precisión, y causan daño si no hay objetos que se interpongan entre el jugador y el enemigo.
    - Mutant: dispara misiles en el mapa.

- Item:
  Clase abstracta de la cual heredan todos los items del juego, hereda de Posicionable. Su único método use() permite a los jugadores usar los items del mapa. Las clases heredadas de Ítem son:
    - Food, Medicine y Blood, estas clases aumentan la vida del jugador;
    - Bullets recarga las balas;
    - Key permite levantar llaves;
    - MachineGun, FireCanon y RocketLauncher permiten adquirir nuevas armas;
    - Treasure, que a su vez heredan de esta clase Cross, Trophie, Chest, Crown, aumentan el puntaje del jugador.

- Object:
  Representa un objeto del mapa del server, hereda de Posicionable. Estos objetos no pueden ser atravesados en el mapa del server a menos que se indique lo contrario.

- Passage:
  Representa un pasadizo en el mapa del juego, hereda de Object. Actúa en el juego como una pared, sin embargo durante el juego se puede remover y cruzar a través de esto.

- Door:
  Representa una puerta en el juego, hereda de Object. Los jugadores pueden abrir las puertas y cruzar a través de ellas, unos segundos después se cierran solas.
  La clase KeyDoor hereda de Door, representando las puertas que necesitan llave para abrirlas. En la primera apertura de esta puerta se necesita una llave, luego actúa de la misma forma que Door.

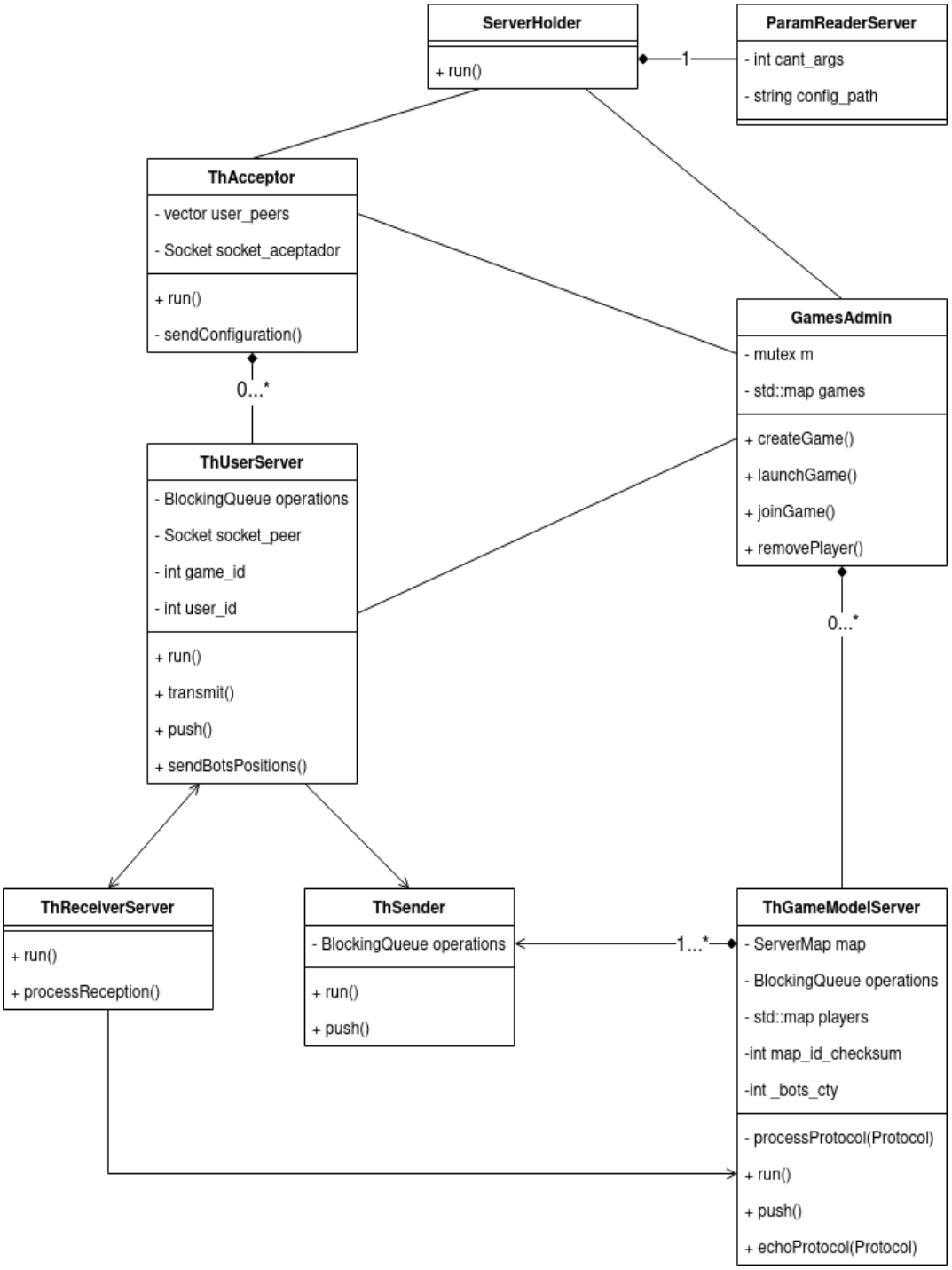
- Rocket:
  Representa un misil del lanzacohetes, hereda de Object. Este misil se mueve en linea recta en el mapa, explotando si colisiona contra algún objeto o jugador. También explota si el jugador intenta moverse hacia la posición en la cual está el misil.

- Event:
  Clase abstracta de la cual heredan FinishGameEvent, DoorOpeningEvent, DoorEvent y RocketEvent. Su único método process() procesa el evento permitiendo enviar protocolos al GameModelServer.
- FinishGameEvent: Controla el tiempo de la partida, envía un protocolo al finalizar el tiempo.
- DoorOpeningEvent: Controla el tiempo entre que una puerta comienza a abrirse hasta que queda completamente abierta, envía un protocolo para que los jugadores puedan cruzarla.
- DoorEvent: Controla el tiempo en que una puerta permanece abierta, envía un protocolo para cerrarla.
- RocketEvent: Controla los movimientos de un misil desde que se lanza hasta que explota, envía protocolos para mover el misil cada cierto tiempo.

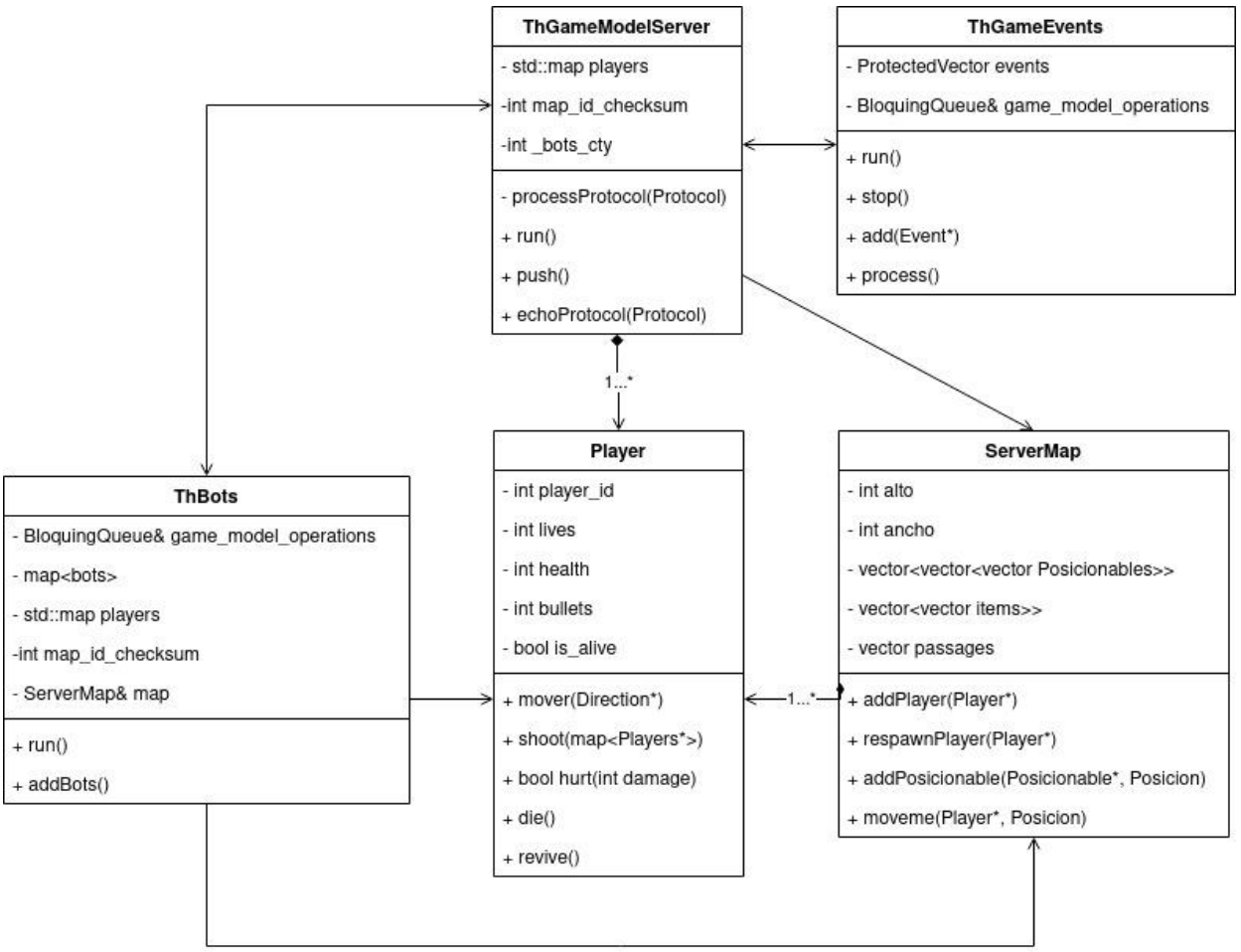
- Bot:
  Representa a un jugador controlado por el juego. Decide las acciones que debe realizar el jugador mediante un script de Lua, estas acciones pueden ser: moverse hacia adelante, girar a la derecha o izquierda, disparar, cambiar su arma o abrir una puerta. Para funcionar necesita información acerca del mapa del servidor, el jugador al cual representa y el resto de los jugadores.
  El script de Lua tiene como objetivo atacar a su enemigo más cercano. Para esto contiene una matriz representando el mapa y un grafo representando los movimientos posibles dentro del mapa. Con esta información y la del resto de los jugadores el script puede decidir qué hacer. Mediante un recorrido bfs (Breadth First Search) del grafo el jugador encuentra a su enemigo más cercano, lo persigue, si tiene que cruzar alguna puerta la abre, y cuando está lo suficientemente cerca lo ataca.

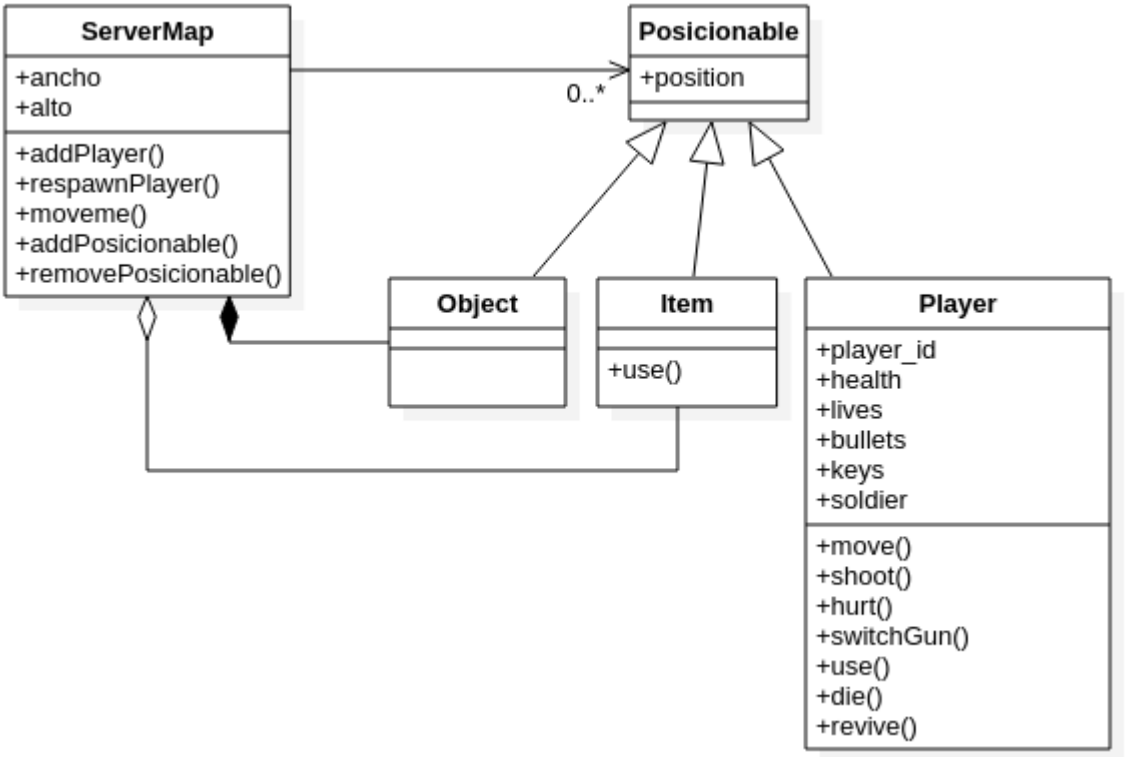
# Diagramas UML

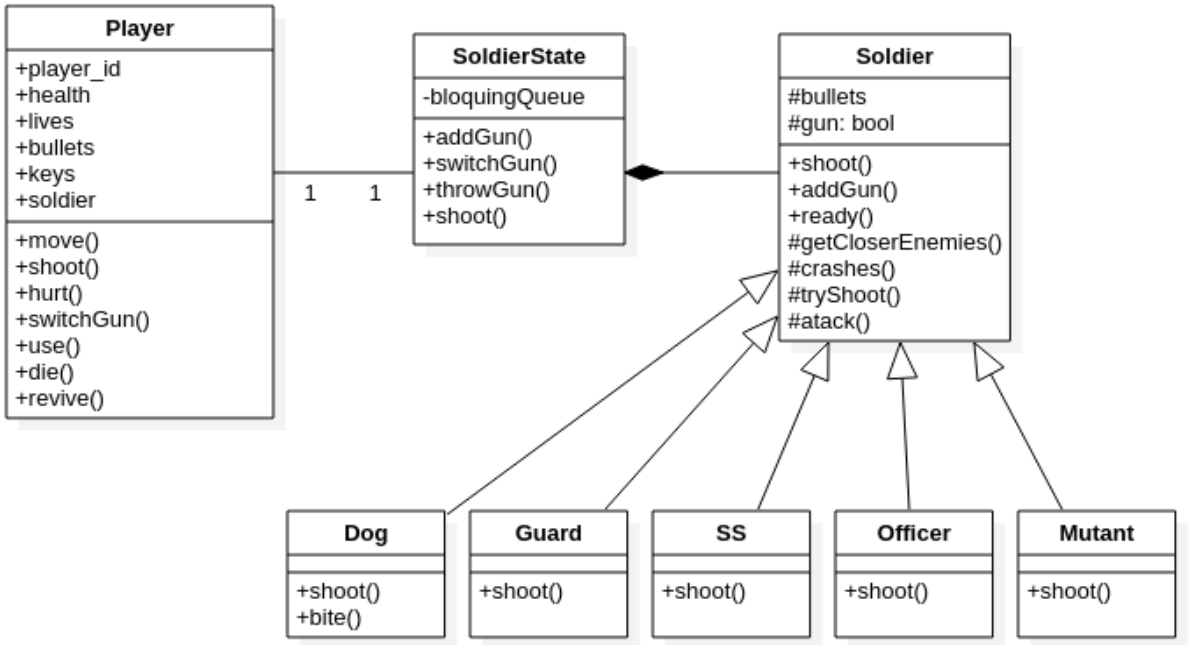
Diagrama: Comunicación con clientes y creación de partidas

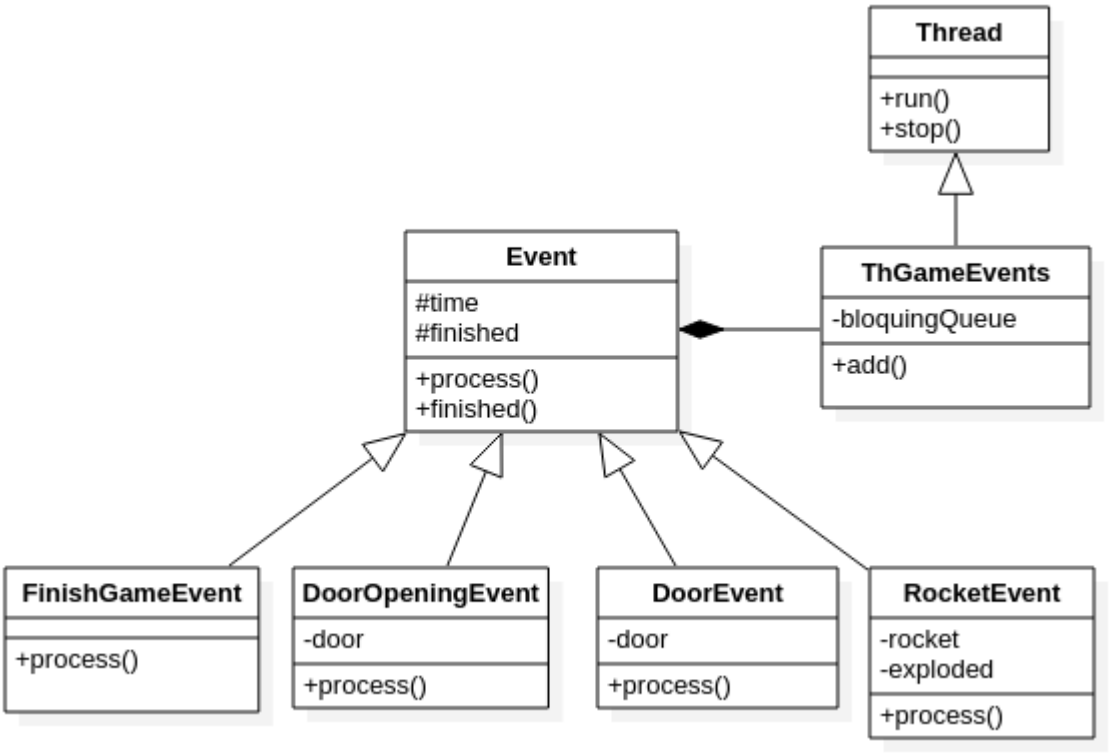


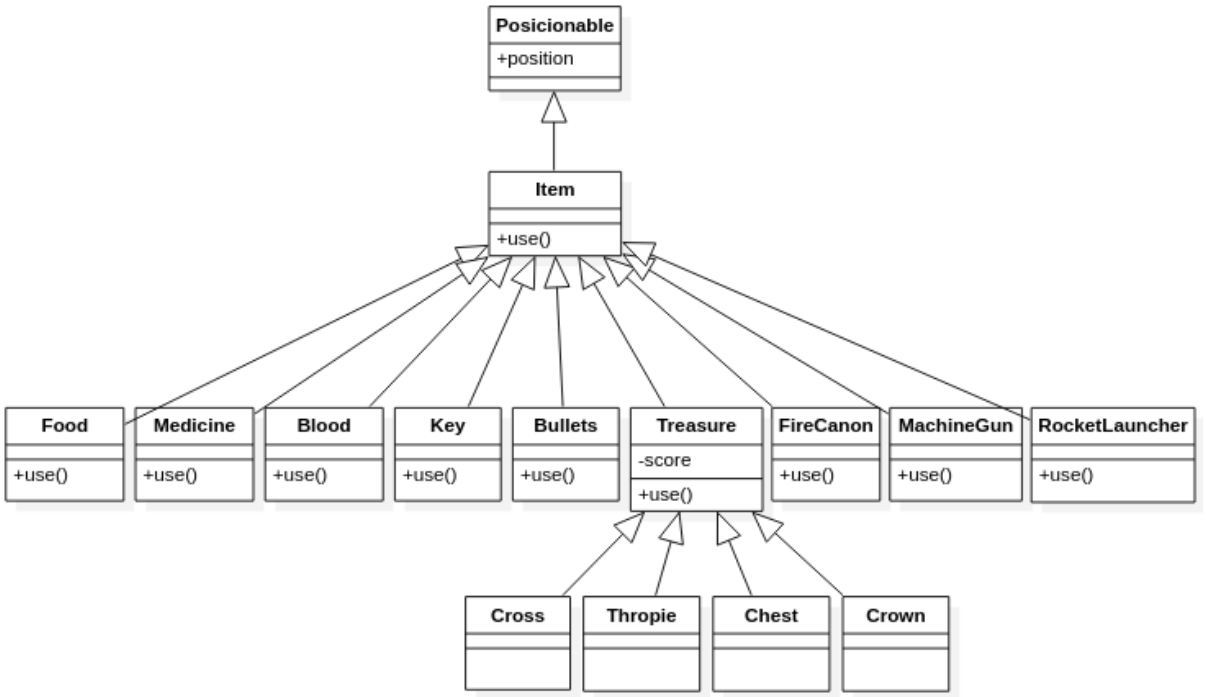
ThGameModelServer:
Internamente cada ThGameModelServer lanza 2 hilos más: Un ThBots y un ThGameEvents, a su vez, el ThGameModelServer se encarga de destruir estos hilos que lanzó.

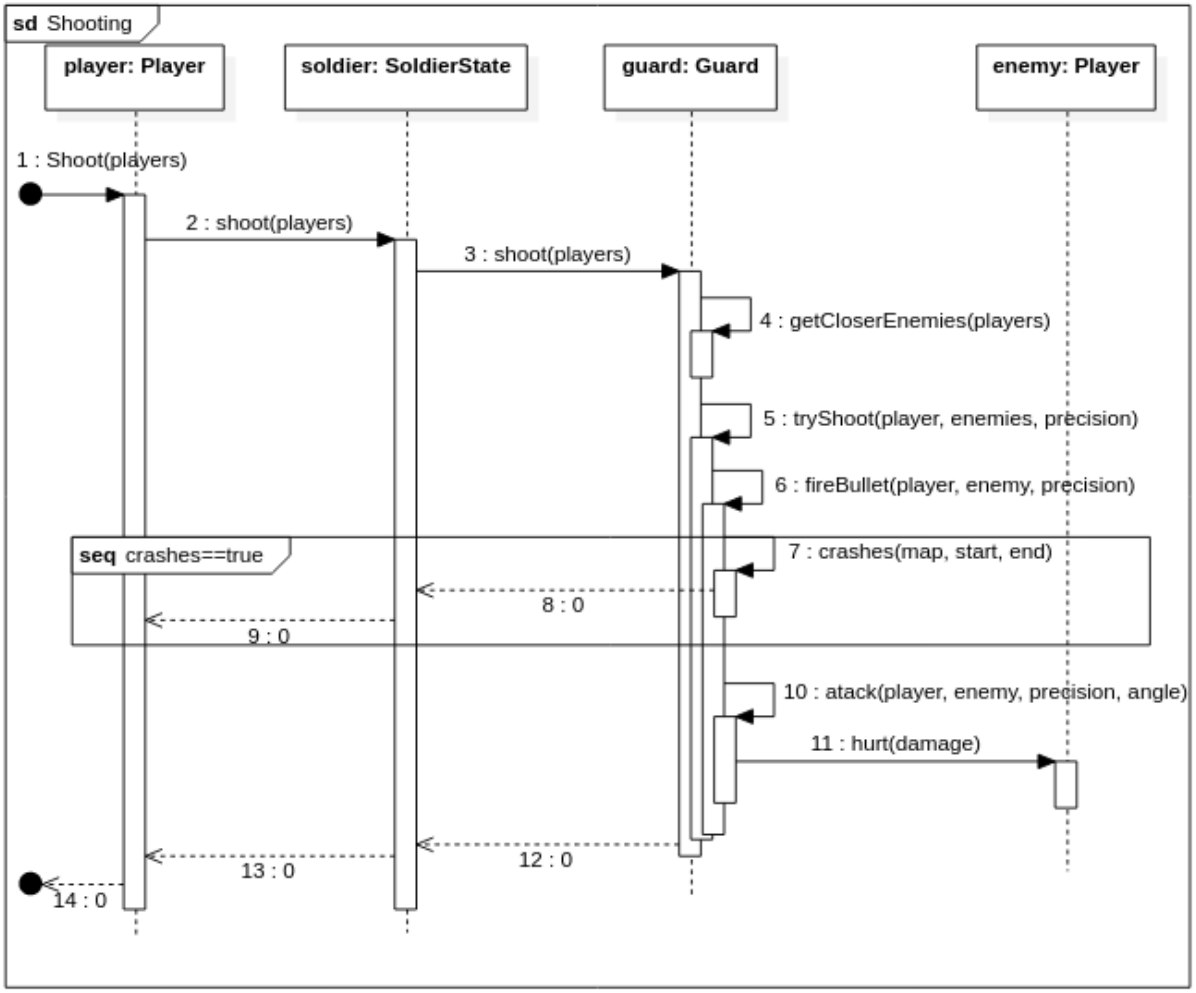



ServerMap
+ancho
+alto
+addPlayer()
+respawnPlayer()
+moveme()
+addPosicionable()
+removePosicionable()
0..*
Posicionable
+position
Object
Item
+use()
Player
+player_id
+health
+lives
+bullets
+keys
+soldier
+move()
+shoot()
+hurt()
+switchGun()
+use()
+die()
+revive()


Player
+player_id
+health
+lives
+bullets
+keys
+soldier
+move()
+shoot()
+hurt()
+switchGun()
+use()
+die()
+revive()
1
1
SoldierState
-bloquingQueue
+addGun()
+switchGun()
+throwGun()
+shoot()
Soldier
#bullets
#gun: bool
+shoot()
+addGun()
+ready()
#getCloserEnemies()
#crashes()
#tryShoot()
#atack()
Dog
+shoot()
+bite()
Guard
+shoot()
SS
+shoot()
Officer
+shoot()
Mutant
+shoot()


Thread
+run()
+stop()
Event
#time
#finished
+process()
+finished()
ThGameEvents
-bloquingQueue
+add()
FinishGameEvent
+process()
DoorOpeningEvent
-door
+process()
DoorEvent
-door
+process()
RocketEvent
-rocket
-exploded
+process()


Posicionable
+position
Item
+use()
Food
+use()
Medicine
+use()
Blood
+use()
Key
+use()
Bullets
+use()
Treasure
-score
+use()
FireCanon
+use()
MachineGun
+use()
RocketLauncher
+use()
Cross
Thropie
Chest
Crown


sd Shooting
player: Player
soldier: SoldierState
guard: Guard
enemy: Player
1 : Shoot(players)
2 : shoot(players)
3 : shoot(players)
4 : getCloserEnemies(players)
5 : tryShoot(player, enemies, precision)
6 : fireBullet(player, enemy, precision)
seq crashes==true
7 : crashes(map, start, end)
8 : 0
9 : 0
10 : atack(player, enemy, precision, angle)
11 : hurt(damage)
12 : 0
13 : 0
14 : 0

# Descripción de archivos y protocolos

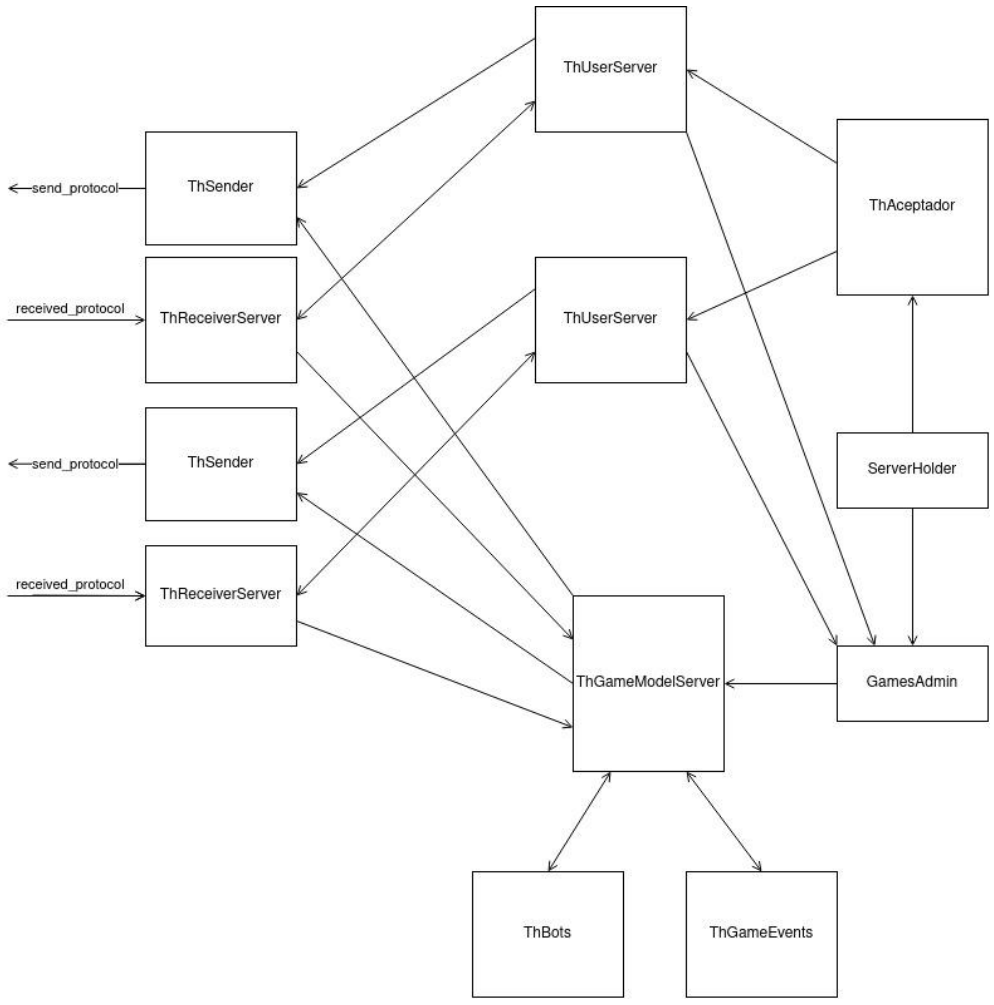
## Archivos

Los archivos utilizados por el servidor son:

- Los mapas ubicados en la carpeta /data/maps
- El archivo de configuración en formato yaml pasado por parámetro al correr el juego, en dicho archivo se establecen valores que favorecen a la jugabilidad del juego y toda esa información es pasada a los programas cliente apenas se conectan al servidor, por red.

## Protocolos

Desde el programa servidor se envían y reciben protocolos por red utilizando sockets TCP. A continuación veremos  la comunicación internamente en el servidor y hacia cada uno de los clientes:




En el previo gráfico podemos observar como la clase principal ServerHolder crea 2 clases ThAcceptor y GamesAdmin, a medida que cada cliente se va conectando al servidor, ThAcceptor va creando nuevos hilos ThUserServer (en este caso a modo de ejemplo 2 clientes se habían unido al juego), los cuales tienen también 2 hilos ThReceiverServer y

ThSender, estos le envían los protocolos que reciben tanto al ThUserServer al momento de crear una partida y al ThGameModelServer al momento de jugar una partida.
Por su parte el ThGameModelServer, recibe protocolos de todos los clientes, los procesa y les contesta una respuesta, ésta clase crea 2 Threads más: ThBots encargado de controlar a los bots de la partida y ThGameEvents, clase que se encarga de manejar todos los eventos que requieran tiempo, una vez procesado, le envía un protocolo al servidor para que este lo procese como cualquier otro y luego les avise a los clientes.

# Módulo Editor

## Descripción general

Para desarrollar el editor, se utilizó QTDesigner. El mismo, nos generó la clase Editor. Esta clase, es la clase principal de la aplicación editor, y la misma se encarga de generar toda la vista como así también manejar la carga y creación del mapa.
Esta clase, contiene una instancia de ResourcesWidget, la cual se encarga de guardar los recursos del juego con los que el usuario podrá generar el mapa, y también, de una primera parte fundamental para el Point & Click.
A su vez, el Editor, contiene un MapWidget, el cual se encarga del ciclo de vida del mapa. El MapWidget, contiene un EditableMap, que contiene toda la lógica relacionada al mapa del juego. La siguiente imagen, muestra el área del cual cada clase se encarga.

# Clases

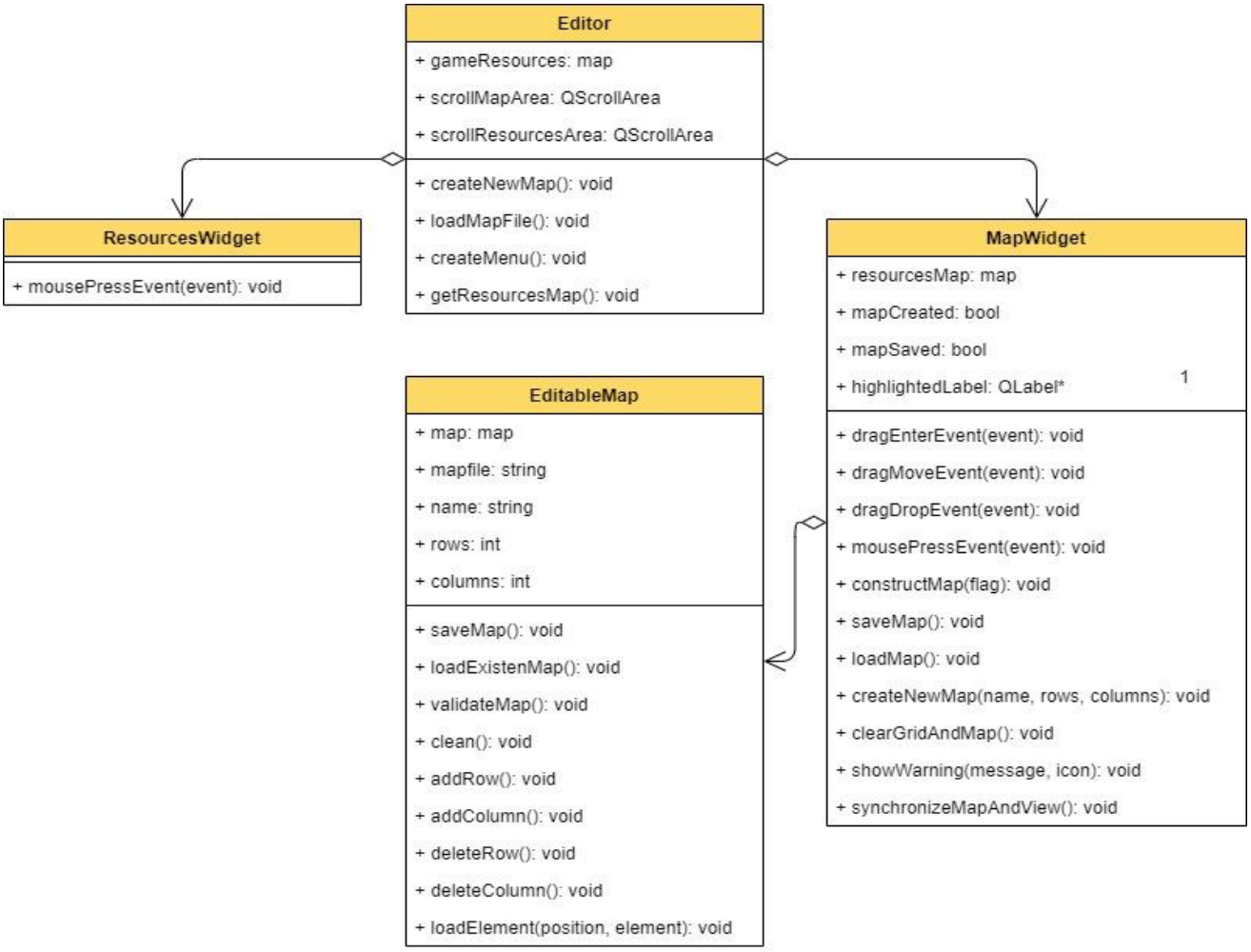
**Editor:** El editor, como mencionamos en la descripción general, es el encargado de generar toda la vista principal y las operaciones principales. Contiene como atributos un MapWidget y un ResourcesWidget, y también todos los recursos del juego a utilizar. Sus funciones son: Comenzar con la creación del mapa, comenzar con la carga de un mapa, crear el menú principal, iniciar el guardado del mapa, detectar si hay cambios sin guardar, agregar la scrollbar al MapWidget como a su vez modificar su estilo, y mostrar diferentes alertas en diferentes casos de uso. Además, esta clase, guarda los distintos recursos de QT que utiliza, como Layouts, Labels y otros.

**ResourcesWidget:** Esta clase, recibe los recursos del juego, y se encarga de graficarlos para que el usuario pueda usarlos para realizar el Point & Click. En esta clase, solo se almacena la responsabilidad de la acción de Point. Además, esta clase, guarda distintos recursos de QT que utiliza como layouts.

**MapWidget:** Esta clase, es la encargada de graficar el mapa y de todo su ciclo de vida. Para graficar el mapa, utiliza una QGridLayout, en la que guarda por cada posición, dos elementos: uno encargado de la visualización del mismo, y el otro contenedor de la información del elemento que guarda. Recibe órdenes de la clase Editor para crear el mapa, cargar uno nuevo, limpiarlo, entre otras. Tiene la responsabilidad de construir el mapa, ya sea nuevo o cargando desde un archivo, finalizar con el Point & Click (En ResourcesWidget se realiza el primer click de Point, y en esta se realiza el segundo click para pintar el elemento), ejecutar el Drag & Drop, agregar o eliminar filas y columnas, mostrar diferentes alertas, entre otras. Contiene un MapaEditable y lo utiliza para las validaciones relacionadas a la lógica del juego, como también para el guardado del mismo. Antes de realizar cualquier acción que requiera alguna validación, se sincroniza la información del MapWidget con la del MapaEditable.
Para el Drag & Drop, se utilizan varios métodos. Uno encargado de iniciar el Drag, otro de detectar cada movimiento que se realiza y finalmente, otro para detectar el drop final. Estos métodos van moviendo la data desde el inicio hacia el final, pudiendo saber en que posición está el cursor en todo momento.

**EditableMap:** Esta clase, contiene el mapa del juego, y realiza las validaciones correspondientes, entre ellas, validar que el mapa tenga paredes principales, que haya al menos un jugador, que las puertas esten rodeadas de paredes, etc. También esta clase es la encargada de leer los archivos .yaml y cargar la información del mapa, como también el caso inverso, exportar toda la información del mapa a un archivo .yaml.

# Diagramas UML




# Descripción de archivos y protocolos

Este módulo, contiene únicamente los archivos correspondientes a cada clase, y además, un archivo más para manejar las excepciones que puedan ocurrir a lo largo del programa.

# Programas Intermedios y de Prueba

Al momento de realizar las pruebas utilizamos el framework “gtest” provisto por Google para realizar pruebas unitarias en C++.
También utilizamos al inicio del proyecto un programa siguiendo el modelo proxy, en el cual creamos un único programa que contenía tanto al cliente como al servidor donde afinamos el funcionamiento general de ambos programas, para luego si, separarlos en cliente y servidor.

# Código Fuente

La totalidad del código fuente puede encontrarse en:
https://github.com/ramaMont/taller-tp-grupal