FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Documentación técnica

CS2D

[7542] Taller de Programacion I
Primer cuatrimestre 2021

Alumno	Padrón
Santiago Pablo Fernandez Caruso	105267
Federico Elias	96105
Mateo Capón Blanquer	104258

Índice

1. Requerimientos de software

Se requiere un Sistema Operativo Linux. Las dependencias necesarias son las siguientes.

1. Compilador g++.
2. cmake.
3. libsdl2-dev.
4. libsdl2-image-dev.
5. libsdl2-ttf-dev.
6. libsdl2-mixer-dev.
7. libsdl2-gfx-dev.
8. libyaml-cpp-dev.
9. qt5.

2. Descripción general

El proyecto cuenta con dos módulos principales: **cliente** y **servidor**. Cada modulo cuenta con su propia lógica y se comunican a través de un protocolo.

El **cliente** es el encargado de manejar la interacción con el usuario (*front-end*), comunica los eventos ocurridos a la vez que procesa las acciones realizadas por el mismo.

El **servidor** es el encargado de manejar toda la lógica del juego (*back-end*), respondiendo de forma adecuada a las acciones que realiza el usuario.

Además, se cuenta con un **Editor de Mapas**.

3. Módulo Común

3.1. Descripción general

Siendo que muchas clases son compartidas por el cliente y el servidor, decidimos crear el módulo *common*. Éste está dividido en seis carpetas, las cuales explicaremos a continuación

3.2. Protocolo de Comunicación

Las carpetas más interesantes e importantes a analizar de este módulo son las de *Communication* y de *Event*. En estas se define el protocolo de comunicación entre el cliente y el servidor.

Siempre que se quiere enviar un mensaje entre cualquiera de los dos, se debe crear un objeto del tipo Event (o que herede del mismo). Éstos responden al método *get_ msg()* con un vector de caracteres, el cual representa al mensaje que identifica al evento en particular.

Además, la recepción y envío de mensajes se genera a través del objeto instancia de *Protocol*. El mismo recibe siempre por referencia un Socket en sus métodos. Si se trata de un envío, recibe un mensaje producido por un evento, mientras que en el caso de la recepción, retorna un evento.

El protocolo utilizado por *Protocol* se basa en el que debimos implementar para el *TP3*. Se envían dos bytes con el tamaño del mensaje y luego el mensaje en sí. Una vez recibido el mensaje, se crea un objeto del tipo *Event* el cual será manejado de manera correspondiente por el cliente o el servidor.

A continuación mostramos dos diagramas de secuencias representando el envío desde el cliente y la recepción en el servidor de un mensaje arbitrario.

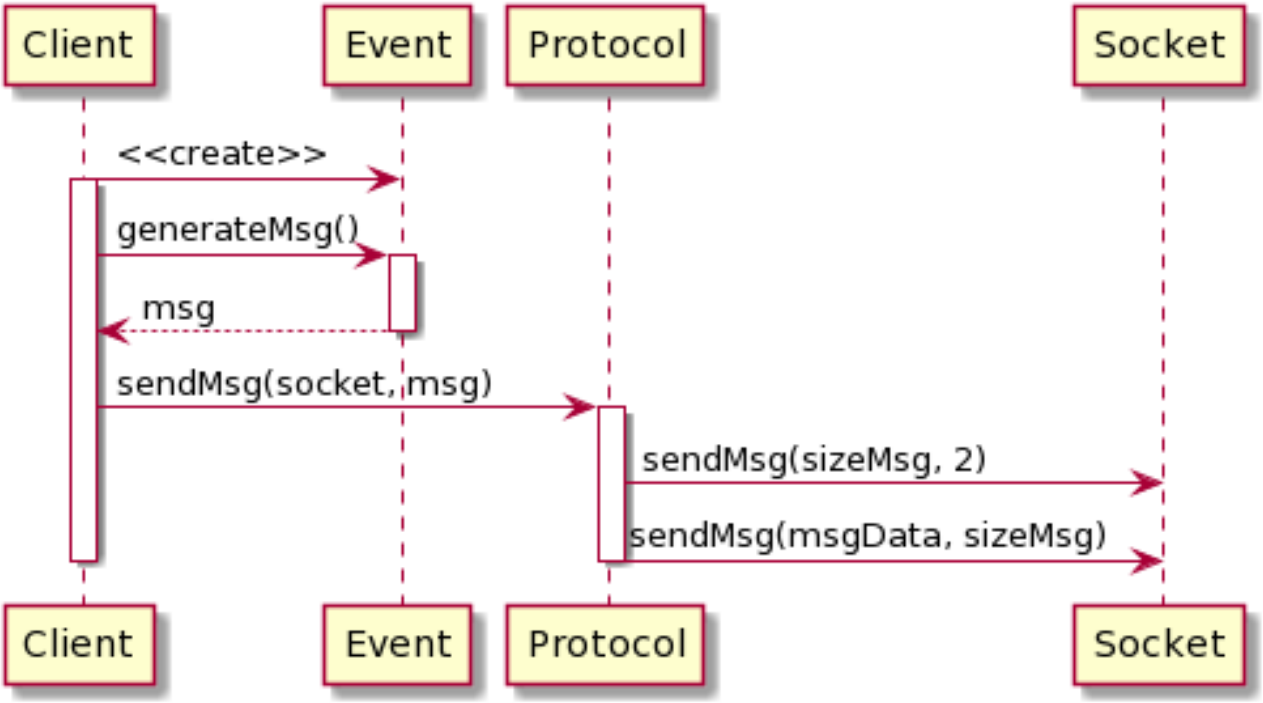

Figura 1: Diagrama de Secuencias del Envío de Mensajes

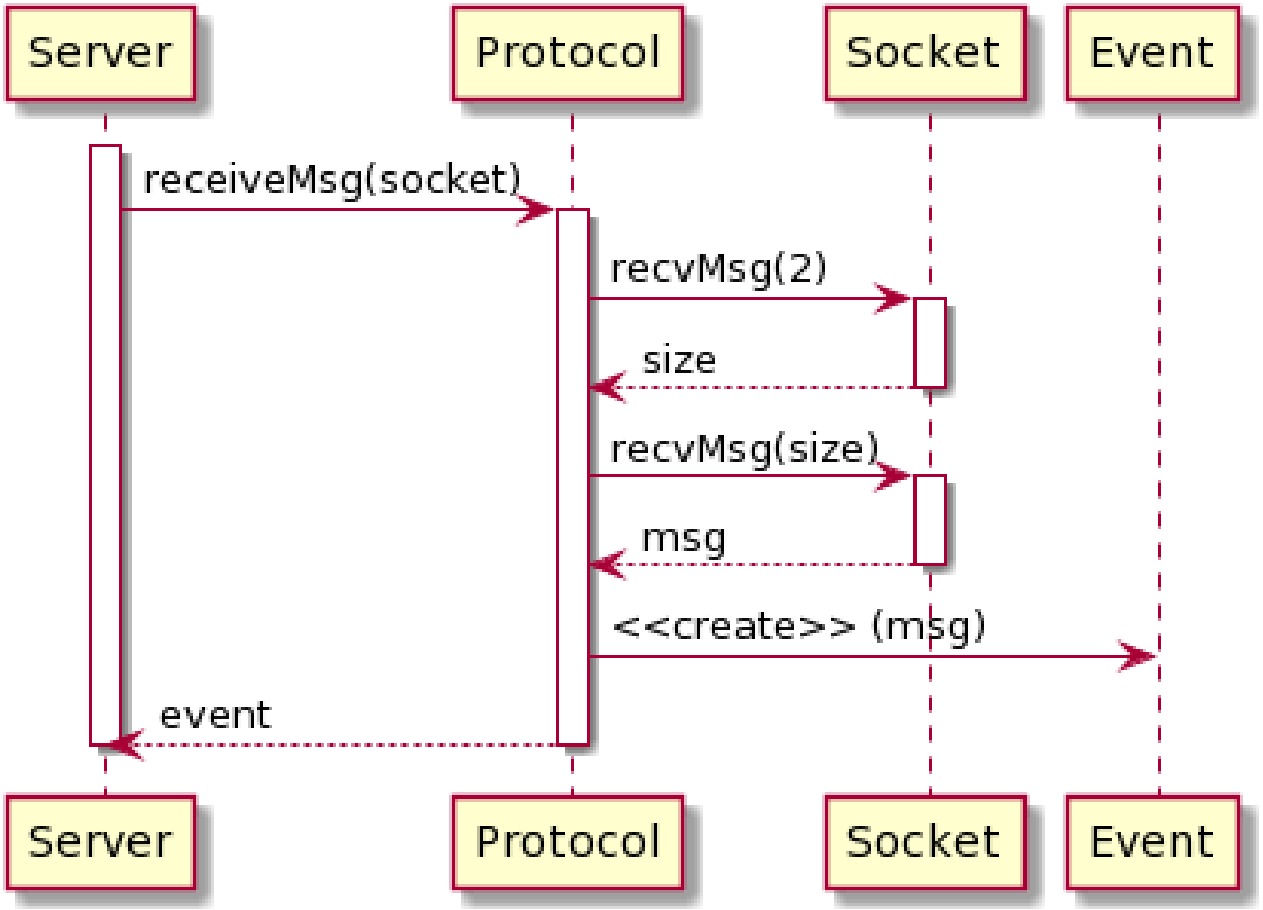

Figura 2: Diagrama de Secuencias de la Recepción de Mensajes

3.2.1. La clase Event y los EventHandler

Todos los eventos tienen un tipo. Todos los tipos de eventos posibles están enumerados en el archivo *TypesOfEvents.h*. Aquellos enviados por el server son los *ServerTypeEvent* y los enviados por el cliente son los *ClientTypeEvent*.

Cada uno de estos eventos tiene asociada una clase que hereda de *Event*, la cual tiene la responsabilidad de poder crear el mensaje correspondiente a su evento, para ser enviada por protocolo. A su vez, cada evento de un módulo (servidor o cliente) tiene un *Handler* asociado en el otro módulo. Éstos deben poder interpretar el evento y manejarlo de manera apropiada. El presente diagrama de clases muestra estas relaciones de un modo muy abstracto.

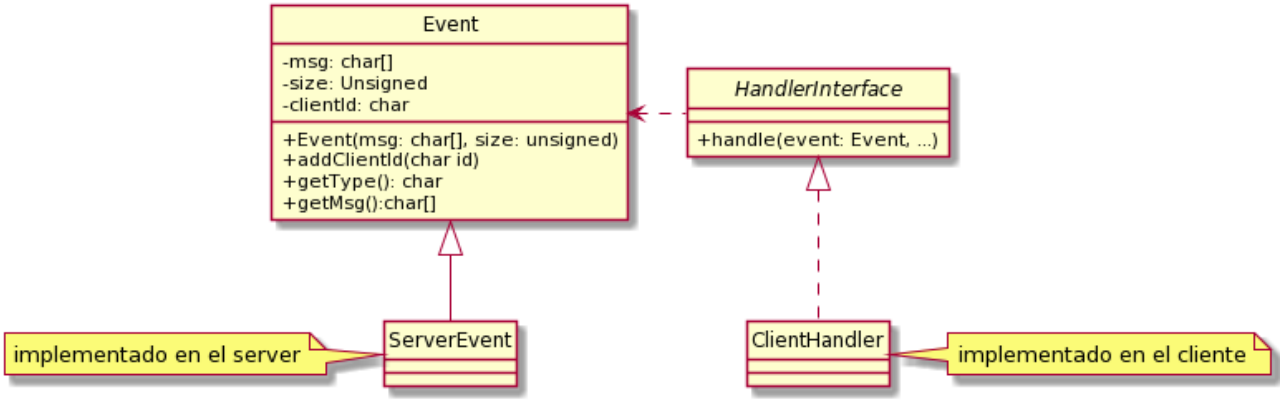

Figura 3: Diagrama de Clases de la relación entre los *Handler* y los *Event*

Estas relaciones nos permitieron separar en módulos claros tanto al cliente como al servidor. Quedaron definidos de este modo las carpetas *ClientEvents* y *ServerEventHandlers* para el cliente; y *ServerEvents* y *ClientEventHandlers* para el servidor. Siempre que se pudo, se buscó esta dualidad entre el cliente y el servidor.

Creemos adecuado aclarar que estos modelos no se respetaron siempre en la práctica. A pesar de que nos hubiese gustado hacerlo, en la etapa inicial del cliente, la comunicación y los manejadores de eventos están todos acoplados en la misma clase: *mainwindow*. Somos conscientes de que no es la manera ideal, pero por falta de tiempos no llegamos a liberar de tantas responsabilidades a esta clase.

3.2.2. Comunicación Sincrónica y Asincrónica

Una forma de dividir la comunicación entre el cliente y el servidor es en la que ocurre en la etapa de Login, y la del loop principal del juego. La primera, sincrónica y la segunda, asincrónica.

De este modo, las carpetas respectivas de *Events* y *EventHandlers* se subdividen en Lobby y Game, siguiendo con la dualidad cliente-servidor ya mencionada.

La inicialización del juego se representada claramente por una comunicación secuencial. Por ejemplo, el cliente envía el mensaje pidiendo unirse a una partida, y a continuación el servidor responde a ese mensaje indicando si puede o no hacerlo.

En contraposición, la comunicación durante el gameloop debe ser asincrónica. A pesar de que el usuario no realice ninguna acción, el juego avanza en su reloj. Este hecho nos forzó a implementar dos hilos de comunicación en cada una de las aplicaciones para cada uno de los clientes en ejecución. Durante la etapa de juego, éstos cuatro hilos se encargan de la recepción y el envío de mensajes entre las aplicaciones. Nuevamente, quedan definidas dos carpetas duales en el cliente y en el servidor referidas a la comunicación entre estos hilos.

3.2.3. La Cola Bloqueante

En la carpeta *DataStructures* se encuentra un sólo archivo que es una clase template de una cola bloqueante.

La misma se caracteriza por tener tanto un pop bloqueante, como un pop no bloqueante. Esta decisión se puede justificar con la cola de eventos de una partida recibidos del servidor desde el cliente. Por ejemplo, para la etapa inicial, donde se debe dar inicio al juego, se espera a que el creador envíe el evento correspondiente. Por lo tanto, se usa un pop bloqueante. En contraposición, dentro del *gameloop* se utiliza siempre un pop no bloqueante, debido a que el tiempo del juego debe continuar a pesar de que no haya eventos a procesar.

3.3. La ruta de los archivos: modo Desarrollador y modo Usuario

La carpeta *Paths*, está compuesta por un único archivo: *paths.h* En éste se definen todos las rutas utilizadas por el cliente y el servidor. Las mismas dependen del modo en el que se compila la aplicación. Las rutas son distintas en modo usuario (los archivos son los que se ubican en la carpeta *build*) a las rutas en modo desarrollador, las cuales son locales a los archivos.

Necesitamos utilizar compilación condicional, y además se debe ejecutar el cmake con un parámetro para que se active el modo USR. Está línea se puede observar en el instalador.

3.4. La carpeta Thread

Todos los hilos implementados en la aplicación heredan de la clase *Thread*, la cual se encuentra en esta carpeta. La implementación de la misma se apoya en la compartida por la cátedra en la clase correspondiente a hilos.

3.5. Excepciones

De modo similar, todas las excepciones heredan de la clase *Exception* implementada en la carpeta con el mismo nombre. En la misma se encuentra una excepción específica: *ExceptionInvalidCommand*. La misma es lanzada siempre que un cliente intenta realizar una operación invalida. La particularidad de esta excepción es que se guarda el tipo de error que tiene el cliente, para que el servidor pueda generar un evento aclarando el error que se produjo.

4. Módulo Cliente

4.1. Descripción General

La aplicación del cliente se encarga de gestionar los eventos del usuario, comunicárselos al servidor y mostrar las respuestas del servidor de forma gráfica.

Tal como fue mencionado en el módulo común, se puede dividir al cliente en una etapa de inicialización y en otra etapa juego.

Para la parte gráfica de la etapa de inicialización se utilizo la librería Qt5 mientras que para la etapa del juego se utilizo SDL2.

4.2. Inicialización

Al ejecutar el cliente, este se encarga de crear el **Protocol**, el **Socket**, los hilos **EventSenderThread** y **ModelReceiverThread**, el **Initiator** y el **Game** (no se implementan los métodos de estas ultimas dos en el diagrama para no entorpecer la lectura).

Como se puede ver, el client crea los threads y les asigna una referencia a un socket, también conocen al protocolo y tienen referencias a las colas de model_evets y client_events , luego, tanto el initiator como el game van a tener una referencia a estos threads, pero también a a las colas de eventos.

El client llama al método *launch()* del initiator y una vez el cliente se conecto a una partida y ésta comienza, el cliente llama al *execute()* del Game.

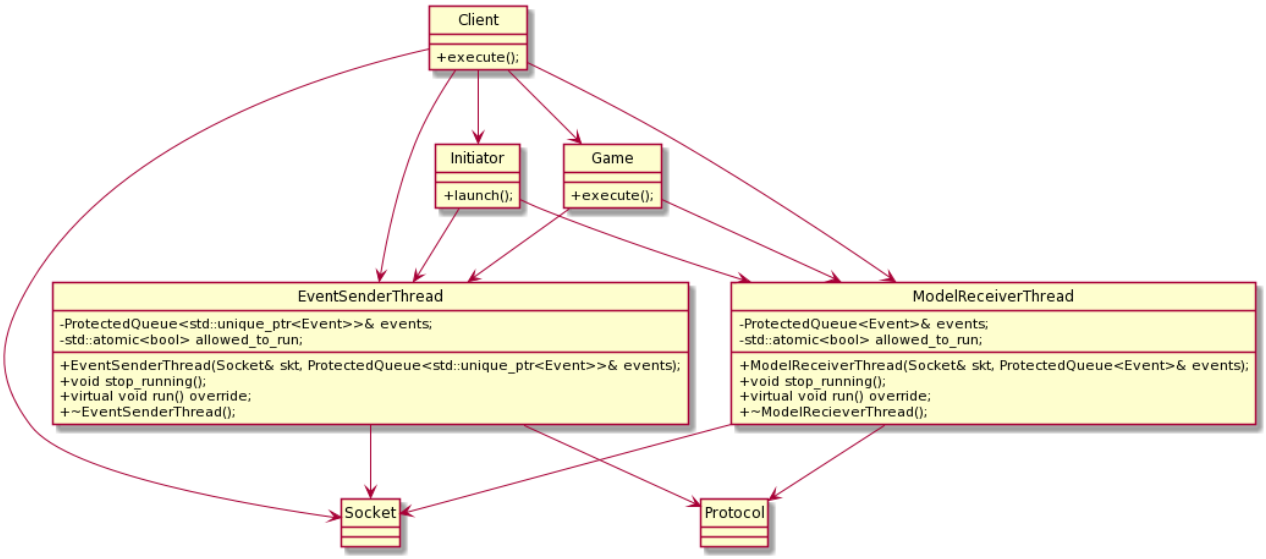

Figura 4: Inicializacion del cliente

4.3. Initiator

Cuando el usuario abre la aplicación del cliente, se abre primero primero la ventada de Initiator, implementada con qt5. Para modelarla, al archivo de ui se le asignaron 7 ventanas:

- **connectToSocket**: Ventana donde el usuario ingresa el host y puerto donde se quiere conectar, en caso de ingresar alguno incorrecto, le saltará un mensaje de aviso y se le permitirá volver a intentar. Una vez ingresados el host y puerto correctos, se conectará al socket del servidor y podrá recibir los datos correspondientes en cada una de las ventanas que siguen.
- **insertName**: Ventana donde el usuario ingresará su nombre. Una vez ingresado, se mandara el mismo al servidor a través de un mensaje, donde será guardado para su uso en las ventanas de espera y en el renderizado de los resultados finales.
- **chooseSkins**: Ventana donde el usuario elegirá su skin de terrorista y su skin de counter, las cuales serán usadas durante todo el juego según el equipo correspondiente. Una vez elegidos ambos se guardan en la configuración del mismo cliente, para luego ser renderizados por éste durante el juego.
- **createOrJoin**: Ventana donde se elige entre crear una partida nueva o unirse a una ya creada.
- **createWindow**: Ventana donde se crea una partida eligiendo alguno de los mapas disponibles. El servidor manda por mensaje el listado de nombres de archivos de los mapas, los cuales son representados en esta ventana como botones. Una vez elegido un mapa se pide que se ingrese un nombre para la partida creada, luego tanto el nombre de la partida como el mapa elegido son enviados por mensaje al servidor.
- **joinWindow**: Ventana donde se unirá a una partida ya creada que no haya empezado. Se recibe por mensaje desde el servidor el listado de partidas disponibles, y se representan cada una con un botón, una vez presionado uno se manda un mensaje al servidor con la partida elegida.
- **waitingForCreator**: Ventana donde el usuario que se une a una partida espera a que el creador de comienzo a la misma. Se reciben por mensaje desde el servidor el listado de jugadores en espera.

- **waitingForJoiners**: Ventana donde el creador de la partida espera a que se unan jugadores, hasa que decide dar comienzo a la misma. Se reciben por mensaje desde el servidor el listado de jugadores en espera, cuando se presiona el botón de comienzo de partida, se manda un mensaje al servidor para que el resto de jugadores deje de esperar.

4.4. Juego

Una vez que el usuario logra conectarse al servidor y entrar o unirse a una partida, comienza la vista del juego en si. Para modelar el juego del lado del cliente se utilizaron tres hilos de ejecución:

- **ModelRecieverThread:** Es el hilo que se encarga de recibir, mediante el protocolo, los *Event* que envia el servidor. Al recibir los eventos se encarga de encolarlos en una *ProtectedQueue* de *Events*.
- **EventSenderThread:** Es el encargado de desencolar los *Events* de la *ProtectedQueue* de eventos de cliente y enviarlos al servidor mediante el protocolo.
- **Hilo principal:** Es donde corre el GameLoop principal, se encarga del procesamiento y renderizado, se utiliza la libreria SDL2 para representar graficamente la informacion del juego.

4.5. La clase principal: Game.cpp

Es la clase principal del cliente en donde se ejecuta el gameloop, se procesan los eventos del usuario y se llama a renderizar lo necesario.

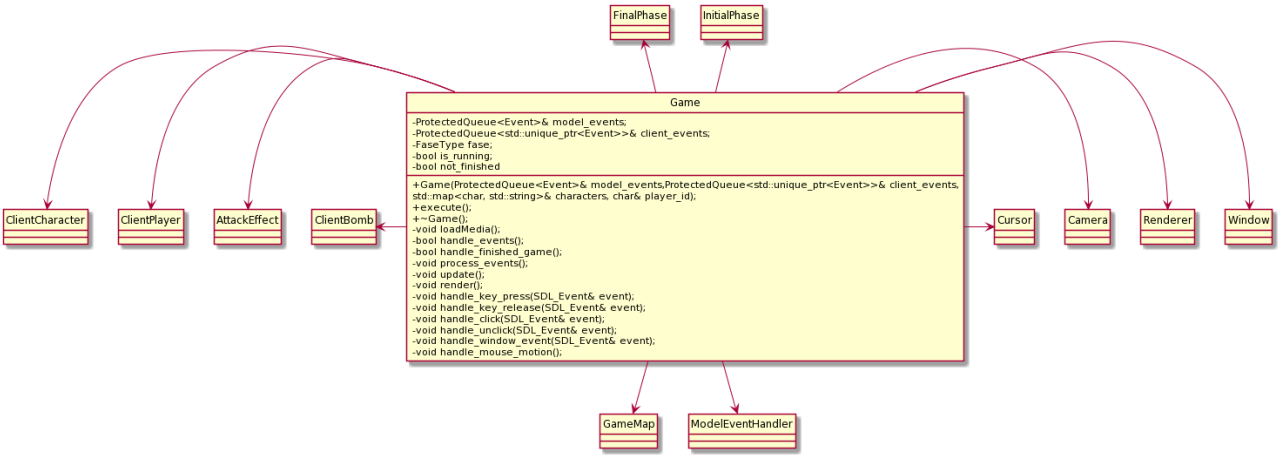

Figura 5: Clase Game

4.5.1. Explicacion de las demas Clases

Dado que un diagrama completo ocupa mucho espacio se explicara el funcionamiento de las distintas clases por separado.

- **GameMap:** Es la segunda clase mas importante del modelo. Es responsable de renderizar el mapa y lar armas y tambien hace de intermediario para poder actualizar los parametros de los distintos jugadores (vida, posicion, etc), del hud y de la camara ,
- **ModelEventHandler:** Clase que conoce a todos los handlers, dependiendo del evento que recive determina que handler lo debe manejar y se lo delega.
- **InitialPhase:** Se encarga de mostrar la etapa de compras y de manejar los eventos de SDL propios de dicha etapa.
- **FinalPhase:** Muestra los stats de las rondas.
- **ClientPlayer:** Representa al player, guarda todos los atributos importantes que son presentados al usuario. Ademas de la textura del jugador tiene una textura para el stencil y otra para el arma que esta portando.
- **ClientCharacter:** Muy similar a ClietPlayer pero estos representan a los demás usuarios.
- **ClientBomb:** Se encarga de todos los featrures de la bomba, muestra las barras de progreso al activar/-desactivar, la animación de la explosion y cuando la bomba no tiene owner la renderiza en el ground.
- **AttackEffect:** Renderiza el efecto de sangre al recibir un ataque.

4.6. Iteracion del juego

El GameLoop del lado del cliente consta de tres pasos principales:

1. Se procesan los **eventos de SDL**, de dónde obtendremos información sobre el accionar del jugador. El procesamiento consta de identificar que tipo de evento esta ocurriendo y en base a dicho evento crear un **ClientEvent** y encolarlo a la **client_events** queue.
2. Se vacía (si son demasiados eventos hay un limite de iteraciones) **model_events** queue, llamando al **ModelEventHandler** para cada uno de ellos, quien se encarga de procesarlos.
3. Se llama a la funcion render donde se limpia la pantalla y luego llama a renderizar a los objetos que sean necesarios segun la fase actual de juego. Por ultimo presenta lo renderizado en pantalla.

4.7. Renderizado

La etapa de renderizado se comporta de diferentes maneras dependiendo de la fase actual del juego. Para poder renderizar texturas se utiliza la cámara, la cual renderiza con un offset con respecto a la posición real del sprite en el mapa ya que la misma se centra en el jugador cada vez que este se mueve, la cámara hace uso del renderer.

4.7.1. Animaciones

Los casos especiales del renderizado son las animaciones, en el trabajo solo hay dos: la explosión de la bomba y la sangre cuando alguien recibe un ataque. En ambos casos se utilizan *ticks* para contar la cantidad de iteraciones que queremos que dure la animacion. Luego de renderizar el clip correspondiente se incrementa en uno la cantidad de ticks, una vez que la cantidad de ticks alcanza el valor que seteamos de duracion deja de renderizar.

4.8. Carga de assets/media

Dado que la cantidad de imágenes y sprites a usar es considerable, antes de iniciar el loop del juego, el Game delega a cada clase la tarea y cada una se encarga de cargar lo que necesita para su correcto funcionamiento.

```
1  void Game::loadMedia() {
2      cursor.setCrusor(SPRITE_POINTER_PATH, 0xFF, 0x00, 0xFF);
3      map.loadMedia();
4      hud.loadMedia();
5      bomb.loadMedia();
6      attack_effects.loadMedia();
7      initial_phase.loadMedia();
8      final_phase.loadMedia();
9  }
```

4.8.1. Contenedores

Un caso especial en lo que respecta a la carga de media es el caso de los Tiles y los Sprites debido a que en el juego hay muchos objetos distintos que deben mostrar lo mismo a la hora de renderizar. No es factible que para cada uno de ellos debamos cargar una nueva textura.

Un ejemplo son los Tiles del mapa, si para cada bloque de un mismo tipo cargamos una nueva textura resultaría muy ineficiente, por ello implementamos dos contenedores, **TileContainer** y **SpriteContainer** los cuales cargan todas las texturas que pueden llegar a ser usadas y luego los objetos particulares toman una referencia a esa textura. Para lograr esto implementamos el patrón **Singleton** para ambos contenedores.

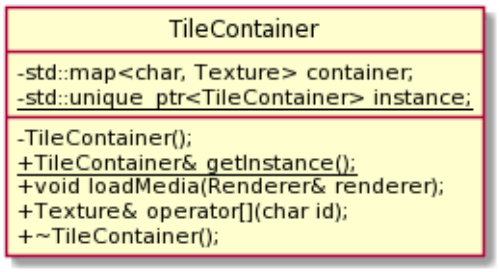

Figura 6: Clase TileContainer

El caso de SpriteContainer es similar.

4.9. Abstracciones de SDL diseñadas

Para utilizar SDL y RAII al mismo tiempo, se diseñaron distintas abstracciones muy utilizadas por todo el cliente. Estas son:

- **Window:** Clase que encapsula la ventana de SDL, la misma ventana se mantiene desde que comienza el Game hasta que el mismo finaliza.
- **Texture:** Clase muy importante que conntiene la propia textura y sus atributos, casi todo lo que se muestra por pantalla son texturas.

- **Renderer:** Encargada de manipular la pantalla, ya sea para borrar su contenido, renderizar nuevos objetos o para presentar la misma al usuario. También se encarga de cargar texturas desde una Surface de SDL. Los métodos más utilizados a lo largo del juego son: clearScreen, render y presentScreen.
- **Cursor:** Permite cambiar estilo del cursor del mouse.

5. Módulo Servidor

5.1. Descripción General

El servidor es el programa en el que se desarrolla la lógica principal del juego. Maneja los pedidos de los clientes que se conectan al mismo y ejecuta el bucle que representa al funcionamiento del Counter Strike.

Tal como fue mencionado en el módulo común, se puede dividir al servidor en una etapa de inicialización y en otra de ejecución del juego.

5.2. Inicialización

Las clases principales cuyas instancias son las encargadas de manejar el login de un cliente, se encuentran en la carpeta *InitialProcess*. En este apartado explicaremos las responsabilidades de cada una de ellas.

Server

El hilo principal de ejecución se encuentra en la clase *Server*. Al llamarse al método *excecute()* se lanza un hilo de una instancia de *Listener* y mientras el mismo se ejecuta, se espera una q por entrada estándar. Una vez que se lee esta letra, se fuerza la interrupción del hilo *escuchador* de clientes y finaliza el programa.

Listener

El hilo escuchador, es el responsable de aceptar a los clientes a través de un *Socket* aceptador. Una vez que se acepta a un cliente, se lo agrega inmediatamente al *Home*.

Home

La instancia de *Home* contiene a todos los clientes en etapa de inicialización. Cada vez que se acepta un cliente, se crea un Lobby, un hilo nuevo de ejecución. Además, se *joinean* a los lobbies ya finalizados.

Teniendo en consideración que los objetos *Lobby* son dinámicos y se encuentran en una estructura de datos, decidimos almacenarlos en el heap. Tal como se menciona en el tutorial interativo de threads otorgado por la cátedra, nos aseguraramos así que no cambiará la dirección de memoria del puntero this. Si esto pasara, la ejecución del thread podría fallar al perder las referencias.

Lobby

La clase *Lobby* se la podría definir como un *EventHandler* que abarca el manejo de todo el conjunto de eventos de Login, el cual los delega a otros *EventHandlers*. Estos son:

1. *CreateMatchHandler*
2. *JoinMatchHandler*
3. *GetMapsHandler*
4. *GetMatchesHandler*
5. *GetUserNameHandler*

A pesar de que en el código no esté incluida la clase *GetUserNameHandler* por falta de tiempo para refinar el código (o elección de otras prioridades), este es un manejador que se adhiere a los otros implementados. Esto es, porque maneja el evento del tipo *ClientTypeEvent::USER_NAME*.

Esta cadena de objetos por los que debe pasar un cliente para ingresarse en una partida la representamos gráficamente a continuación.

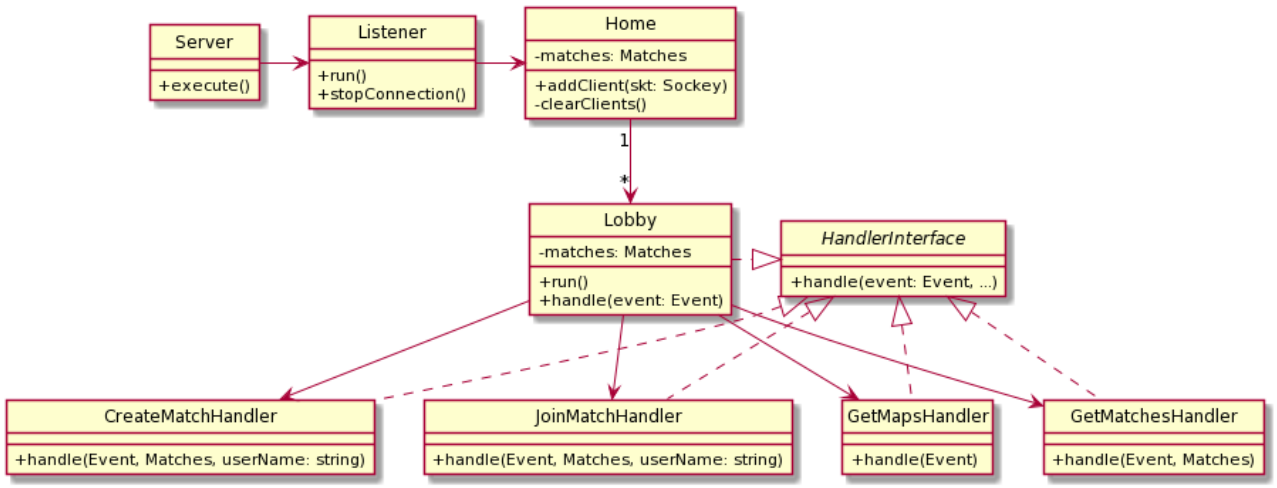

Figura 7: Diagrama de Clases del Proceso Inicial

No lo incluimos en el diagrama para no sobrecargar el mismo de información, pero tanto *Lobby* como *Listener* heredan de *Thread*.

En estos *handlers* mencionados, se implementa la comunicación secuencial que enunciábamos en la sección anterior. Con el objetivo de mostrar cómo se implementan los mensajes sincrónicos, mostramos los diagramas de secuencias correspondientes a los métodos *handle()* de *CreateMatchHandler* (el cual es muy similar al de *JoinMatchHandler*) y de *GetMapsHandler*(el cual se asemeja al de *GetMatchesHandler*).

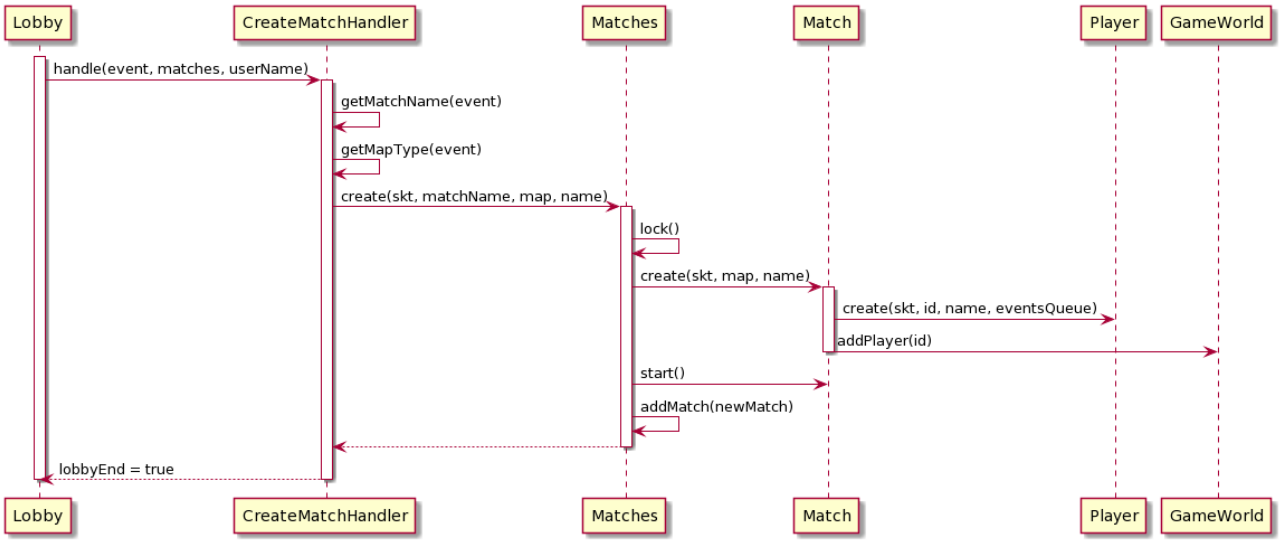

Figura 8: Diagrama de Secuencias del handle() de *CreateMatchHandler*: Parte 1

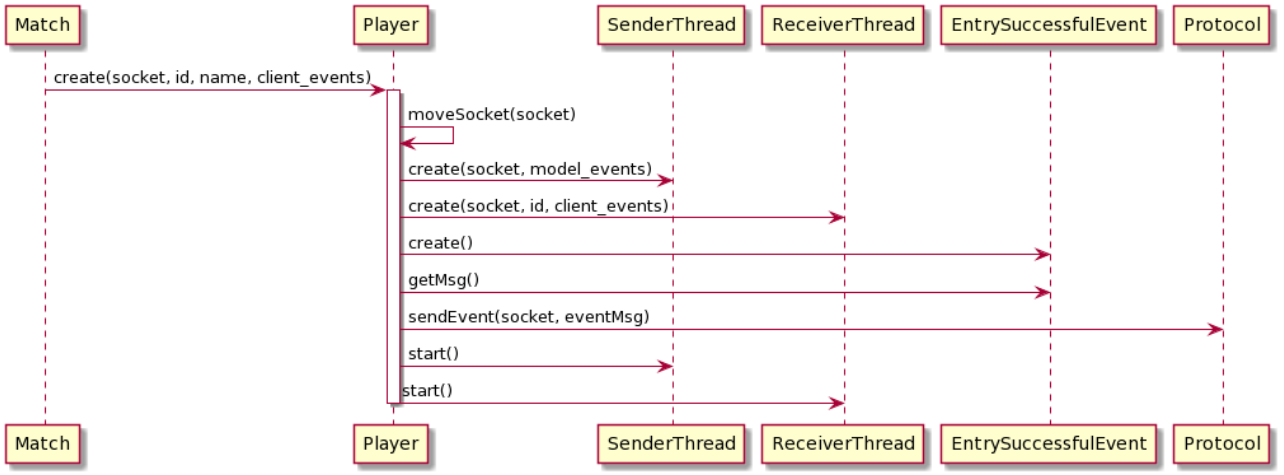

Figura 9: Diagrama de Secuencias del handle() de *CreateMatchHandler*: Parte 2

De este diagrama hay un par de conceptos sobre los cuales nos interesa hacer énfasis. En primer lugar, cuando se intenta crear el *Match*, *Matches* bloquea su mutex. Esto es necesario para evitar una race condition, ya que *Matches* es un recurso compartido por todos los clientes en etapa de Lobby. En segundo plano, se debe observar que al crearse el *Player*, el socket es movido y empieza a pertenecerle. En otras palabras, el lobby (y con este la comunicación sincrónica) se da por finalizado. El último mensaje que pertenece a los mensajes secuenciales es el envío del evento *EntrySuccessfulEvent*, el cual le avisa al cliente que debe también finalizar su etapa de login.

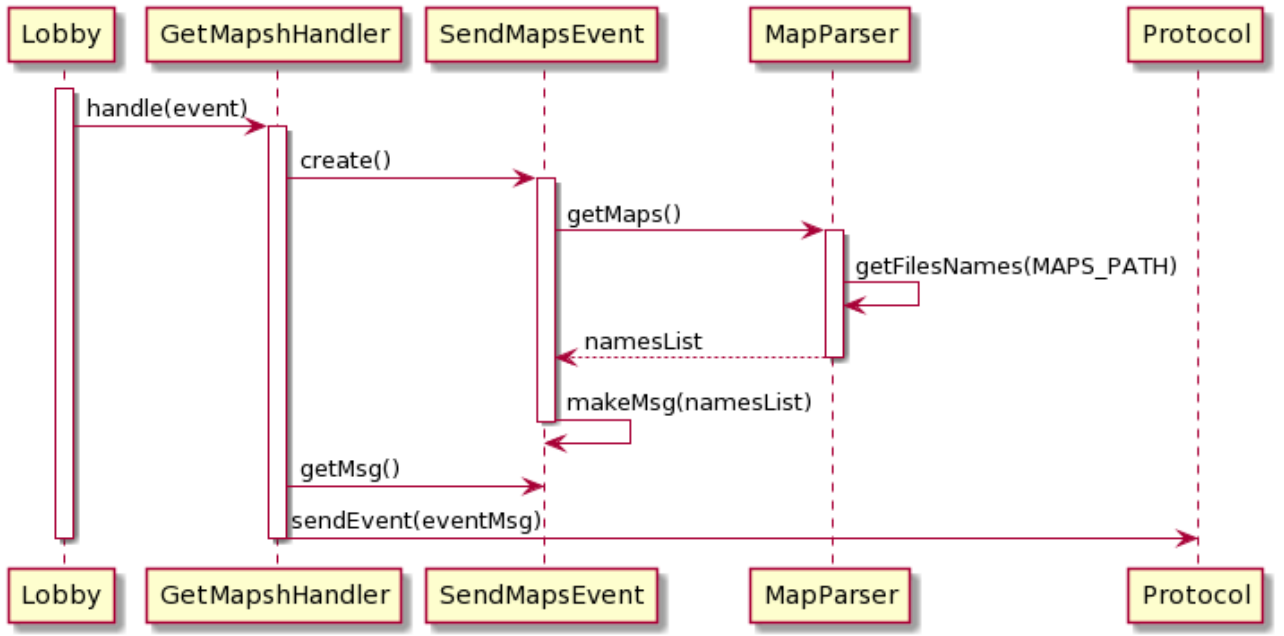

Figura 10: Diagrama de Secuencias del handle() de *GetMapsHandler*

En el presente diagrama buscamos nuevamente ejemplificar la comunicación sincrónica: ante la pregunta por los mapas, se devuelven los mismos. Además, se puede observar claramente cómo cada clase delega responsabilidades en las demás. Esto genera que las clases sean más abiertas a modificaciones. Por el momento, solo se están enviando los nombres de los mapas, pero fácilmente podríamos implementar que se envíe además el contenido de los mapas, así el cliente puede ver una vista general del mapa que está por elegir.

5.3. Juego

La clase que realiza de nexo entre la primera etapa y la segunda, es *Matches*. Tal como fue mostrado con anterioridad, las instancias de *Match* son creadas por *Matches*. Además, para unirse a uno, se debe acceder a través de *Matches*. A continuación mostramos un diagrama de clases que ayudará a la comprensión de la implementación del juego.

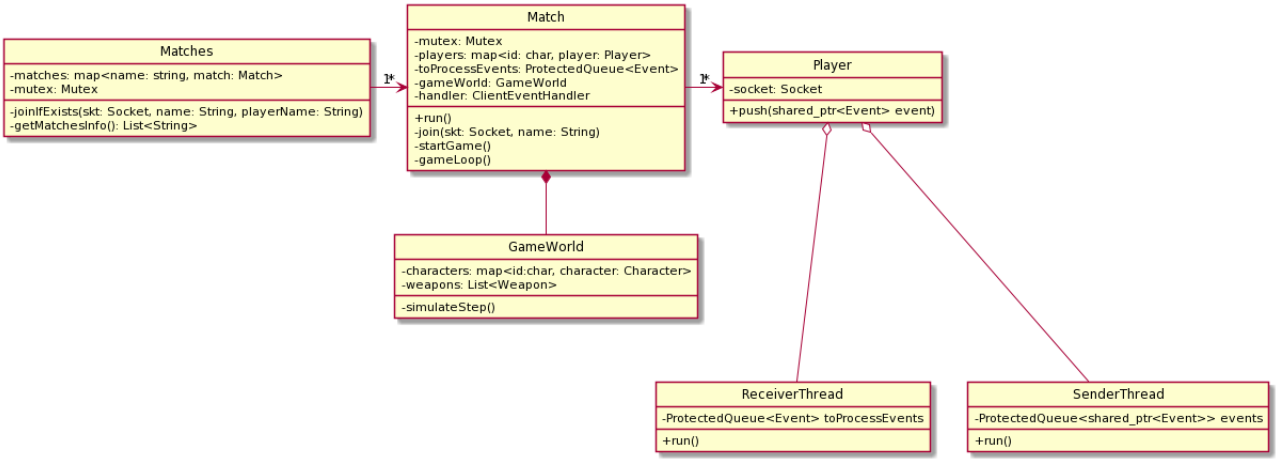

Figura 11: Diagrama de Clases de los *Matches*

En el presente diagrama de clases se presentan las principales clases que manejan la ejecución de las partidas, y la comunicación con los clientes durante las mismas.

Matches

Matches es un contenedor de un conjunto de *Match*. Dado que *Match* es un Thread y además el conjunto de éstos es dinámico, decidimos almacenarlos en el heap. *Matches* es un recurso compartido por todos los clientes en Lobby. Es por esto que está protegida con un mutex.

Match

La clase *Match* también está protegida. Esta decisión la justificaremos con el siguiente ejemplo.

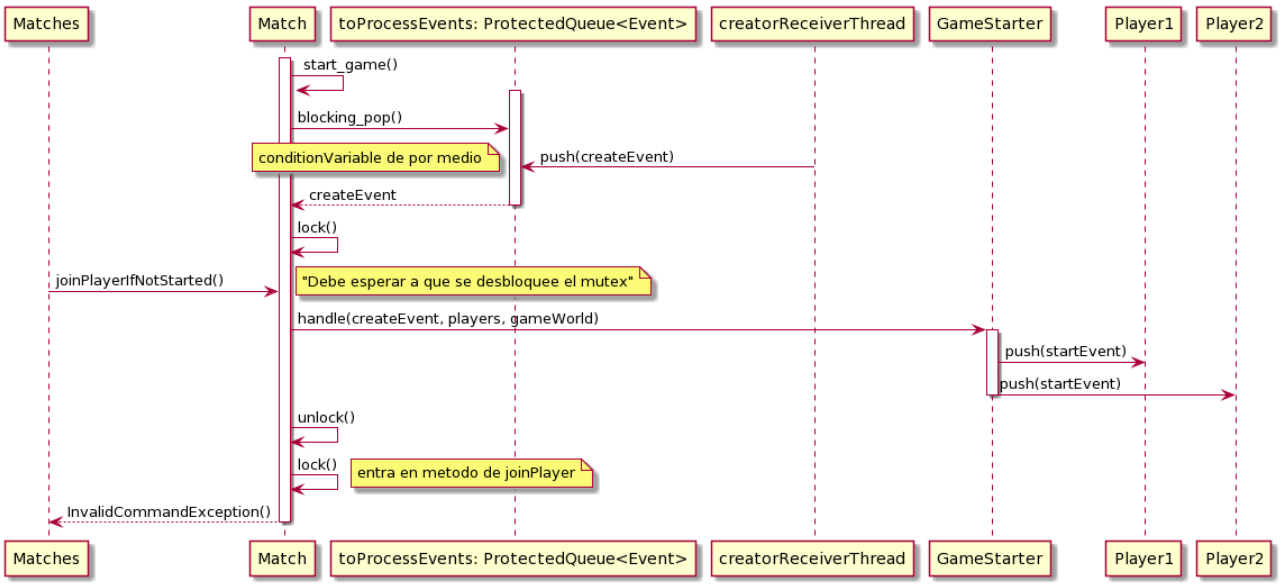

Figura 12: Diagrama de Secuencias del inicio de un *Match*

Una vez que se inicia una partida, todos los players deben ser avisados. En el presente diagrama se observa que si un cliente se intenta unir a una partida que está por comenzar, podrá hacerlo solamente si se une antes de que se bloquee el mutex a causa de que la cola bloqueante recibió el evento de inicio de partida. Desde ya, si esto no sucede, una vez que intenta unirse, no podrá hacerlo, dado que la partida ya está comenzada.

Otra situación clara en la que es necesario el uso de un mutex en esta clase es cuando se cierra el servidor durante el transcurso de una partida. Al cerrarse el servidor, se deben finalizar todos los hilos de los clientes, aquellos que tienen los players. Si se intenta pushear un mensaje a un player al mismo tiempo que se intenta notificar al player que deje de correr, podría llegar a generarse una race condition.

Player, Sender, Receiver y recursos compartidos

Todo *Match* cuenta con dos tipos de colas protegidas. La cola, a la cual tienen una referencia todas las instancias de *ReceiverThread*, solo contiene eventos de la clase padre *Event*. Desde ya es necesario que sea protegida esta cola, ya que todos los hilos receptores pushean en la misma. Durante el gameloop de *Match*, se hacen pop no bloqueantes.

El otro tipo de cola es el que contienen los *SenderThread*. Ésta contiene punteros compartidos de *Event*. En primer lugar, es necesario utilizar punteros, porque los eventos que tiene en la cola heredan de *Event*, y polimorfismo en C++ es con punteros. Segundo, y más importante todavía es el hecho de que todos los hilos envían siempre los mismos mensajes a los distintos clientes. Por lo tanto, todos los hilos pueden manejar los mismos recursos para los mensajes. Es por esto que los punteros son compartidos. Solamente se aloca memoria una vez por mensaje, y se elimina la misma una vez que el último cliente pushee el contenido del mismo. Desde ya, si la conexión entre un cliente y un servidor se pierde, el contador del puntero compartido es restado inmediatamente, al no agregarse nunca a la cola bloqueante, pues nunca se podrá enviar ese mensaje.

La primera cola es no bloqueante, mientras que la segunda es bloqueante. En contraposición, el envío de mensajes a través del protocolo es no-bloqueante, mientras que la recepción de eventos sí los es. De este modo, ambos hilos *SenderThread* y *ReceiverThread* se quedan esperando en una gran porción del tiempo en que están corriendo en sus respectivos métodos bloqueantes, y se evita un bucle difícil de manejar en memoria.

GameWorld

Hasta el momento se explicaron conceptos que podrían ser aplicados en cualquier sistema cliente servidor con múltiples partidas y clientes. La carpeta *GameWorld*, y los handlers que lo conocen se centran en el juego Counter Strike 2D. En estos, se implementa la lógica correspondiente al juego. A continuación mostramos un diagrama de clases sobre la estructura general del mundo.

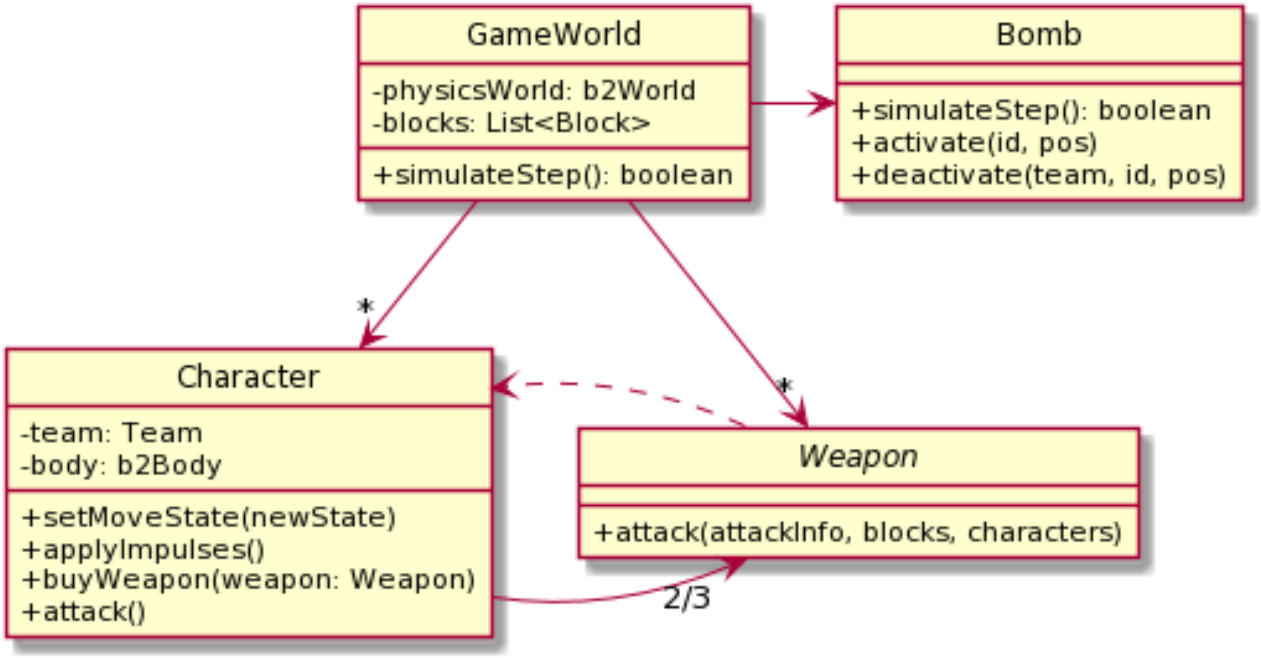

Figura 13: Diagrama de Clases genérico de la carpeta *GameWorld*

En el diagrama no incluimos a muchas otras clases que también tienen sus responsabilidades claras en la carpeta a analizar, para no sobrecargar el mismo de información.

En primer lugar nos interesa destacar el método público de *GameWorld*, *simulateStep()* . En este se simula una unidad de tiempo en el mundo del juego: tanto la sección física, como aquellas partes que no requieren de un modelado físico, pero si deben suceder en cada iteración del bucle del juego. Para la simulación de los eventos físicos se utiliza el *FrameWork Box2D*. Los objetos principales dentro del mundo físico son los bloques (elementos estáticos) y los caracteres (elementos dinámicos).

Por otro lado, en todas las simulaciones del mundo, se debe avanzar un instante de tiempo, STEP_TIME. Es por esto que cuando se llama a la simulación del mundo, también se llama a la simulación de la bomba y de los ataques.

La bomba es un objeto que cambia de estado luego de cierto tiempo si está en modo activando, activada o desactivada. En nuestro proyecto el estado de la bomba viene dado por un atributo enumerativo. Sin embargo, el modelo mas orientado a objetos para la misma hubiese sido utilizando el patrón de diseño *Strategy* o *State*, donde cada vez que se indica que se active o se desactive, cambie su estado interior.

Las instancias de *Character* representan a cada uno de los *Player* mencionados. El *Character* es el jugador dentro de la lógica del juego, mientras que el *Player* representa la comunicación cliente-servidor. En el diagrama de clases mostramos dos de sus tantos métodos: *applyImpulses()* y *setMoveState()*. El primero es el encargado de aplicar los impulsos sobre el cuerpo físico del character. Busca una variación en la cantidad de movimiento del mismo, para que se mantenga constante su velocidad. Por el otro lado, el estado de movimiento le indica en qué dirección debe aplicar este impulso. De este modo queda desacoplada la acción que hará el *MoveHandler*(cambiar el estado) del que hará el mundo al simular una unidad de tiempo(aplicar los impulsos).

Esta separación entre acciones que modifican el estado de los character y otras que se realizan en tiempo de simulación del STEP_TIME, se observa también en la secuencia de los ataques.

Weapon

La clase Weapon es abstracta, pues tiene un método abstracto: *attack()*. Tanto el mundo, como los caracteres tienen punteros a *Weapon*. Esto es así, para poder utilizar el polimorfirmo en C++. Las hijas de Weapon deben implementar el método *attack()*, y por lo tanto son clases concretas (no abstractas). La creación de los Weapon la implementamos sobre la base del patrón de diseño *Factory*. A continuación mostramos el diagrama de clases del mismo.

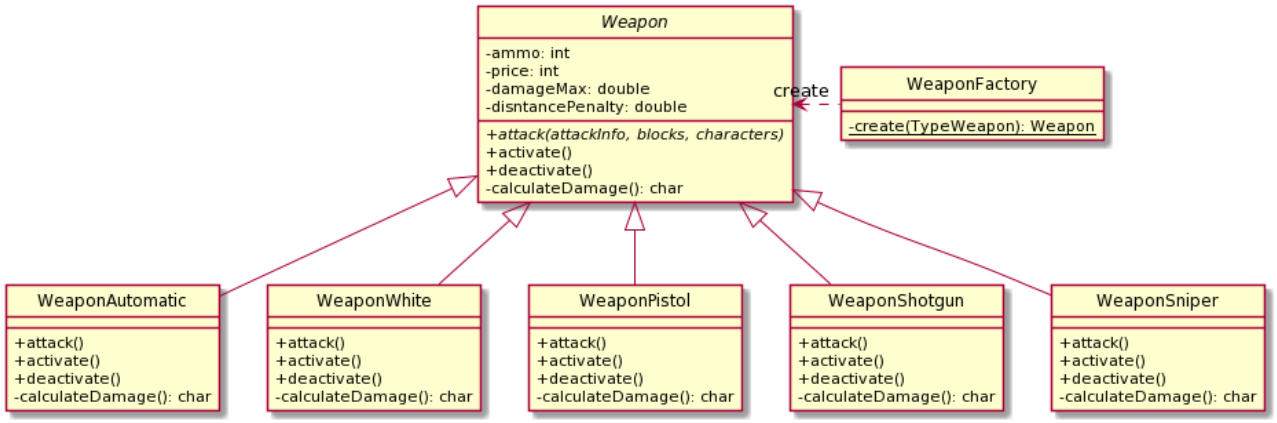

Figura 14: Diagrama de Clases del Factory para *Weapon*

El almacenamiento de las armas, además de ser mediante punteros, es mediante punteros únicos *unique_ptr*. Esto es porque cada arma solo le puede pertenecer a un objeto: o al mundo, o a un jugador. Pero nunca puede ser compartida.

GameLoop

El game loop desde el lado del servidor es muy similar al del cliente. Se basa en los tres pasos principales:

1. Procesar Eventos
2. Actualizar el modelo
3. Enviar Eventos

y un cuarto paso que consiste en un tiempo para descansar al hilo.

Volviendo al ejemplo del movimiento del character, en la etapa de procesamiento de eventos, se le indica al character que cambie su estado de movimiento. Sin embargo, en la etapa de actualización del modelo, se mueve realmente.

De igual modo para los ataques, en la etapa de procesamiento de eventos, se activa el ataque del jugador; y en la etapa de actualización se genera el ataque. Mostramos la secuencia a continuación.

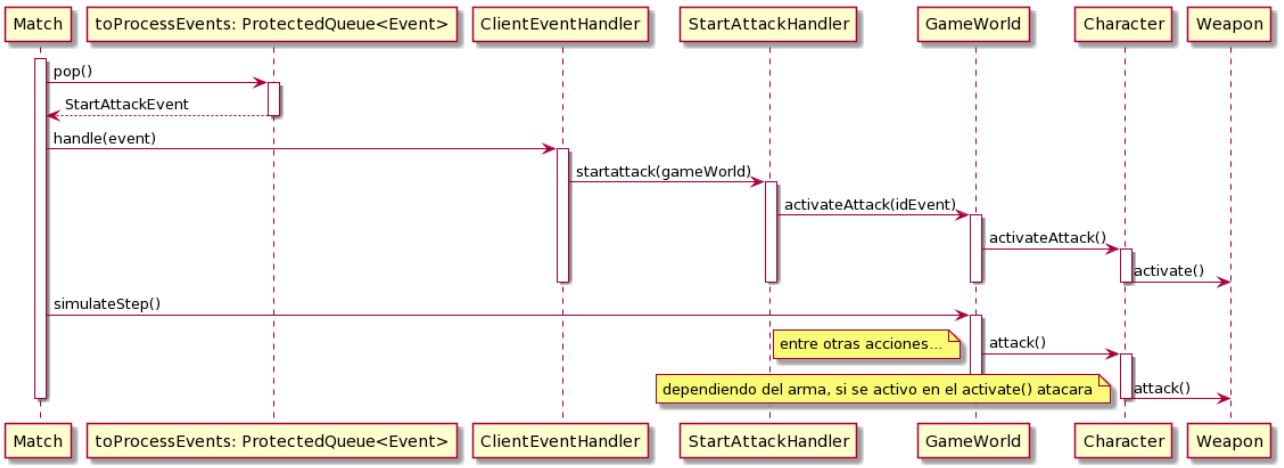

Figura 15: Secuencia del ataque

Para poder procesar los eventos y conocer aquellos aspectos del modelo que se modificaron, se debieron implementar clases que conozcan casi de forma omnisciente a los objetos del juego. La principal es *StepInformation*. Esta conoce tanto a la bomba, como al world, y a todos los caracteres. Es la encargada de generar los mensajes que serán procesados por los *EventHandlers* del módulo del cliente.

6. Módulo Editor

6.1. Descripción general

- **map editor:** es la única clase del Editor, y se encarga de implementar toda la lógica de sus responsabilidades.

  Implementa el mapa sobre el que se colocan los ítems que van a dar forma al mapa del juego, en el cual se además de las distintas acciones que ayudan a la experiencia de usuario, éstos son guardar y cargar el mapa con un archivo de configuración YAML, borrar todos los ítems del mapa, o borrar ítems seleccionados. A la hora de guardar un mapa, recorre todos los ítems del mapa y crea un nodo por cada uno con la información correspondiente: tipo de ítem, coordenada x, coordenada y, y además agrega un nodo que representa el tamaño del mapa.

  Antes de crear el archivo YAML, recorre todos los nodos en busca de huecos dentro del mapa, que rellena con cajas negras para evitar que los jugadores caminen por espacios no renderizados; con el mismo objetivo, también se rodea al mapa con cajas negras, así ningún jugador podrá salirse del mapa. Para evitar rodear de cajas negras cada vez que un mapa se guarda, y así evitar que se agranden los mapas innecesariamente, se agrega un nodo al archivo YAML que indica si este mapa ya fue rodeado de cajas o no, que al cargar el mapa se guarda este valor en un booleano que el editor verificará antes de guardar el mapa.

7. Programas intermedios y de prueba

No se hizo uso de programas intermedios.