Documentación Técnica

Requerimientos de software:

OS: Linux/Ubuntu 18.04 (versión mínima)
Bibliotecas:

- SDL 2
    - SDL_Mixer
    - SDL_TTF
    - SDL_IMG
- MsgPack
- Qt5
- JsonCpp

Herramientas:

1. Compilador: g++ estandar c++11
2. Testing: cppunit
3. Análisis de memoria: Valgrind
4. Depurador: gdb
5. Generación de archivos para compilación :Cmake

Descripción general:

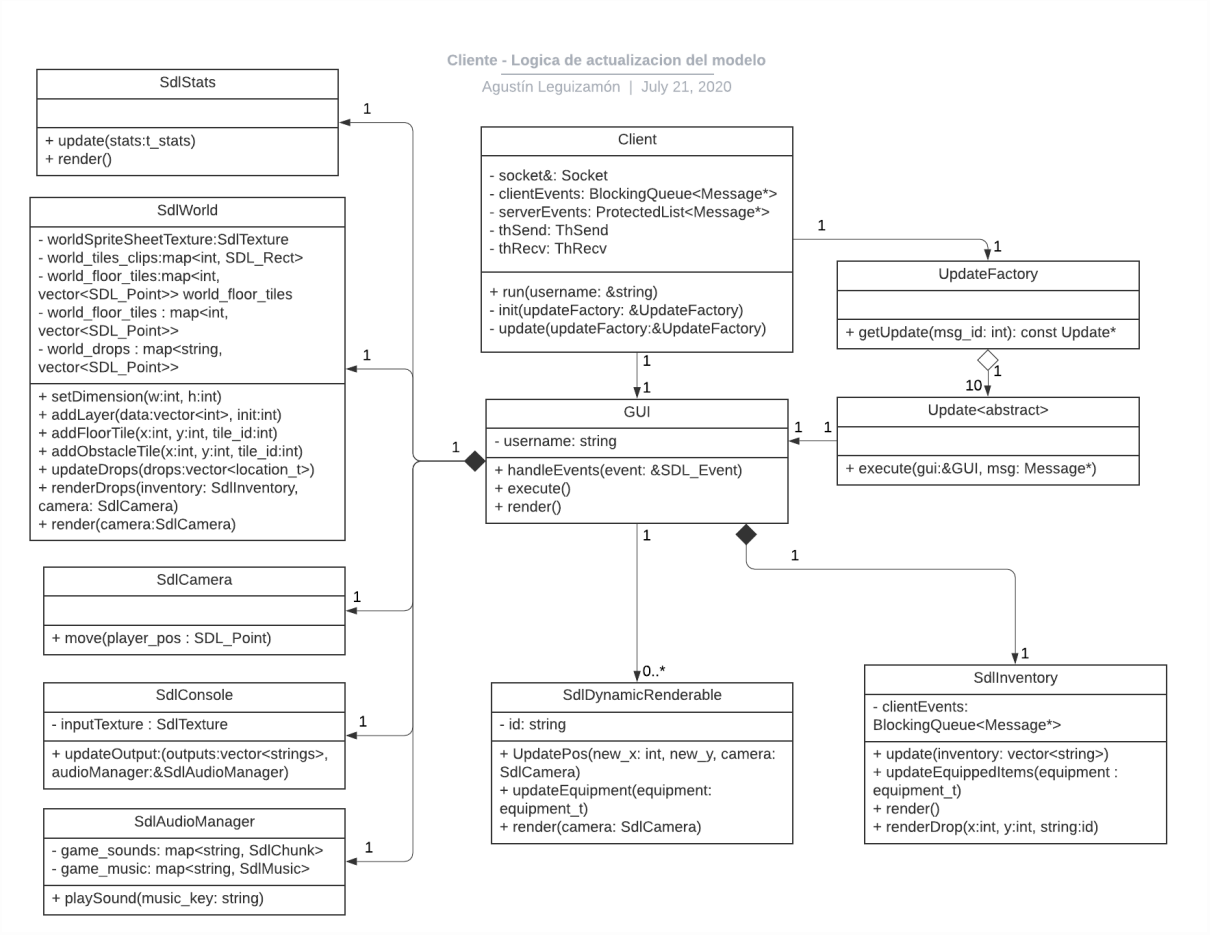
El juego está dividido en dos partes, **Cliente** y **Servidor.** El servidor es el que posee toda la lógica del juego, maneja la interacción entre distintos jugadores y criaturas que conviven en un mismo mundo, además se encarga de guardar los avances de cada cliente para que estos puedan seguir jugando desde donde dejaron. El **Cliente** se encarga de mostrar con gráficos y animaciones lo que sucede en el mundo, jugadores, criaturas, items toda esta información la recibe del **Servidor**. Todo esto se ve renderizado en la pantalla. Además el **Cliente** se encarga de capturar las acciones que hace el usuario a través de la interfaz y se las envía al **Servidor**.

Cliente

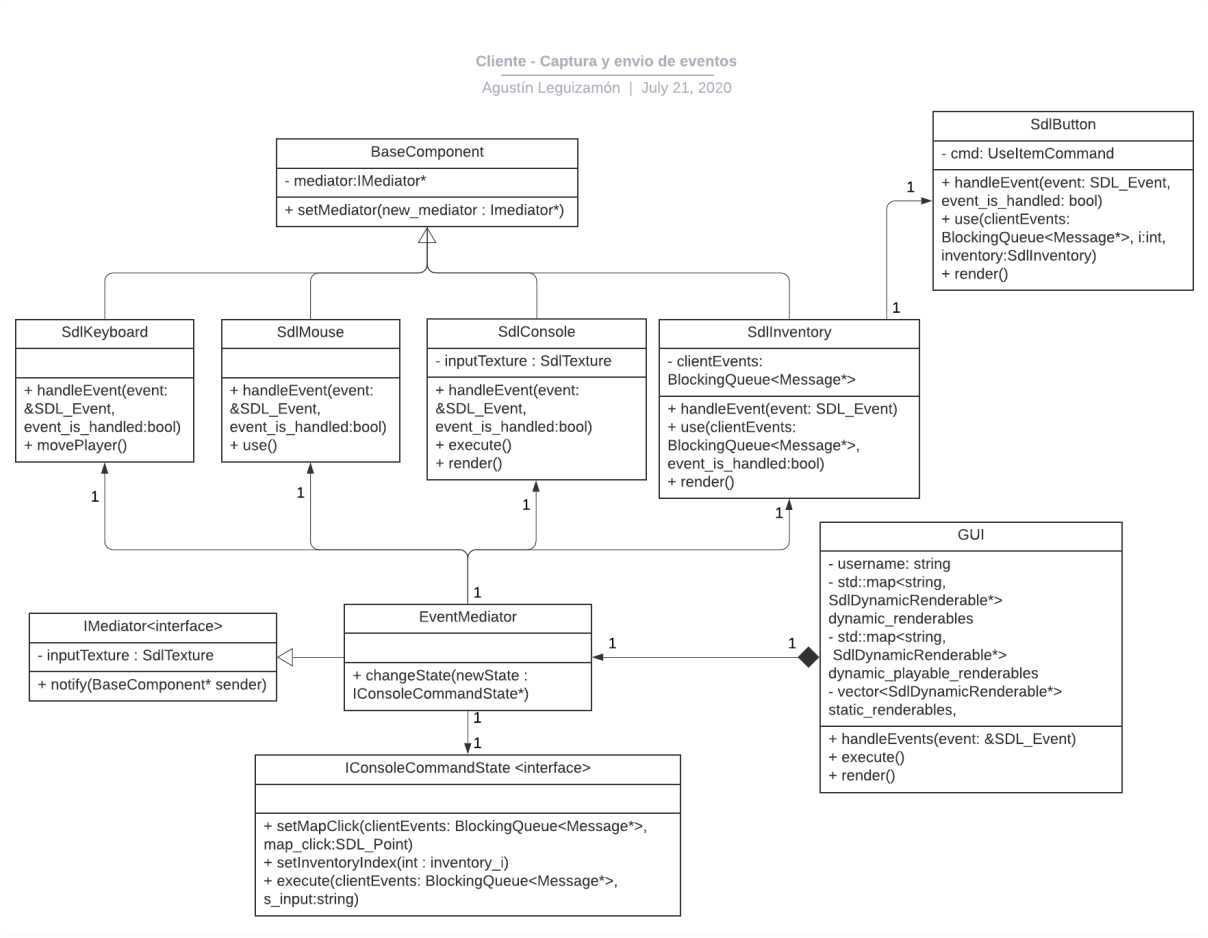
El **Cliente** utiliza dos librerías, Qt5 para el login y SDL para el juego. La clase **GUI** es la base del funcionamiento ya que se encarga de delegar las 3 tareas principales del gameloop (handle, logic y render). La GUI se compone de varias clases (keyboard, mouse, console) a las cuales se les delega las distintas tareas de manejo de eventos y que conocen a una clase EventMediator que utiliza el patrón Mediator para comunicar los eventos entre las clases antes mencionadas y generar el mensaje que será enviado por el socket. Además posee los distintos métodos para la actualización del modelo que recibe por parte del **Server**.

A continuación se muestran dos diagramas de clases con los principales componentes del cliente.

Por un lado tenemos el Factory de actualizaciones que a partir del mensaje del servidor no da uno de los 10 Updates encargados de ejecutar el método del GUI correspondiente.
Los Updates poseen la información de los Mensajes del **Server** para cambiar los atributos de los distintas clases de la GUI (SdlInventory, SdlConsole, SdlStats) ademas se encuentra SdlWorld encargado del renderizado del mapa.



Por el otro lado tenemos las distintas componentes de la interfaz para la interacción con el usuario que capturar los eventos y los procesas a través del EventMediator que se encarga de encolar los Mensajes en la cola protegida ClientEvents para luego ser enviados al server a través del socket. El **EventMediator** permite desacoplar los distintos elementos de la interfaz, estas le notifican cuando procesan los eventos y esto resulta útil en el caso de que haya acciones que dependan de varios elementos, como por ejemplo el hacer click en un NPC para enviarle un comando (interactúan SdlMouse y SdlConsole pero no de manera directa sino a través del EventMediator)



Servidor

Como se mencionó anteriormente el servidor contiene todo lo que se refiere al juego en sí(lógica, información de juego) y además se encarga de mantener a los clientes actualizados acerca de lo que va ocurriendo en el juego.
En la carpeta servidor tenemos dos módulos importantes : **Model** y **Connection**.Como su nombre lo indica en el módulo Model tenemos todo el código referido al modelo y en el de Connection el referido a la conexión del servidor

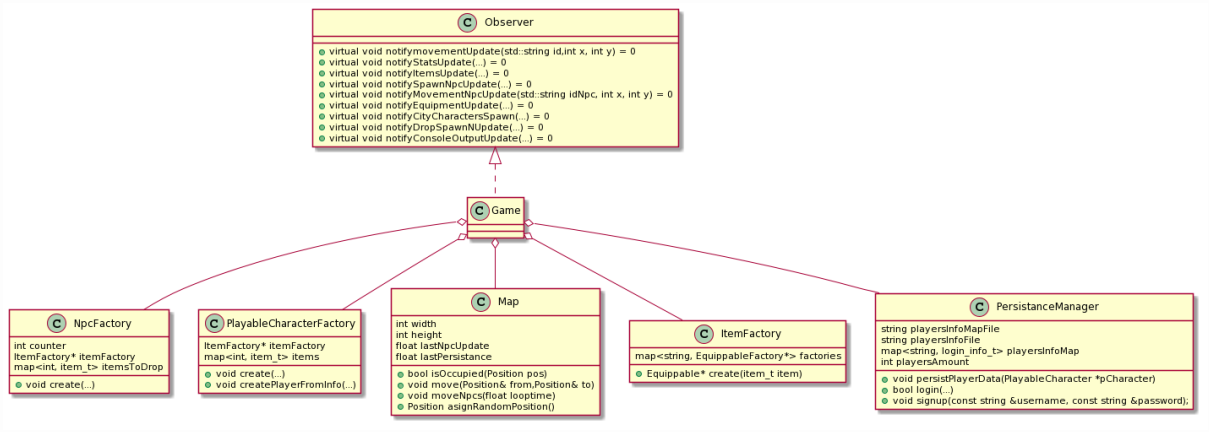
Model

Ahora destacaremos las clases mas “importantes” de este modulo

- Game:
  Esta clase es la principal del modelo, se ocupa de notificar actualizaciones(implementa interfaz Observer) y de delegar tareas a las clases de menor nivel.
  Sus métodos principales son :
  - void initialize() : Se ocupa de inicializar los npcs del juego

- std::queue<Message *> initializeWorld() : Se ocupa de obtener toda la información necesaria que necesita el cliente para inicializarse
- void updateModel(float looptime) : Se ocupa de actualizar el modelo respecto del tiempo transcurrido. Actualiza los stats de los jugadores, mueve los npcs y además en este método es donde se regeneran los npcs.
- Implementa los métodos de la interfaz observer para notificar actualizaciones en el modelo ej :
    - void notifymovementUpdate(std::string id,int x, int y)
    - void notifyDropSpawnNUpdate(...)

Diagrama de clases de la clase Game:



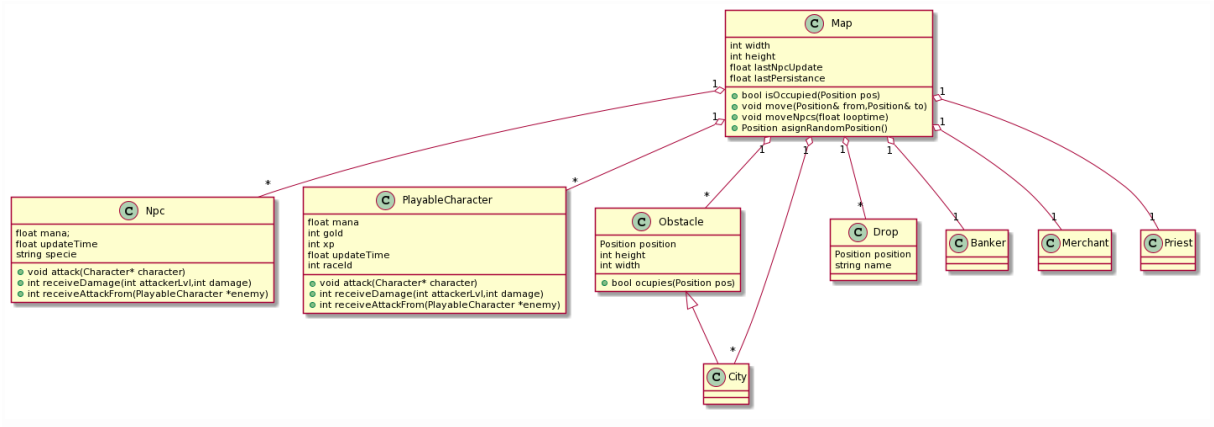
- Map:

Esta clase se ocupa de guardar y administrar las posiciones de todos los personajes y objetos que se encuentran en el mapa.

Sus métodos principales son:

- bool isOccupied(Position pos) : Nos indica si la posición pasada por parametro se encuentra ocupada
- void move(Position& from,Position& to) : Nos permite mover un jugador desde una posición a otra
- void moveNpcs(float looptime) : Ejecuta el movimiento de los npcs
- Position asignRandomPosition() : Devuelve una posición desocupada al azar

Diagrama de clases de la clase Map:



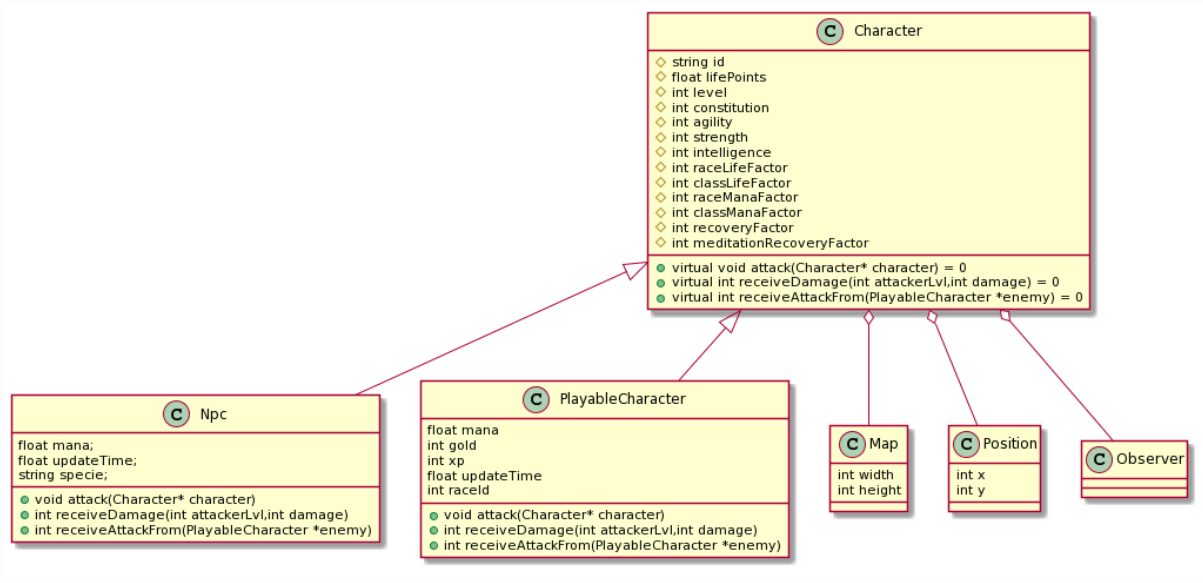
- Character:

Esta clase representa a los personajes activos, es decir, aquellos que se pueden mover,atacar,etc que hay dentro del juego.Aqui se guardan los valores de las habilidades de cada jugador y los factores relacionados a su clase y raza.

Sus métodos principales son:

- virtual void attack(Character* character) = 0
- virtual int receiveDamage(int attackerLvl,int damage) = 0
- virtual int receiveAttackFrom(PlayableCharacter *enemy) = 0
- Además en esta clase se encuentran las ecuaciones que se utilizan para realizar los cálculos del juego(danio,defensa,etc)

Diagrama de Clases de la clase Character:



- PlayableCharacter:

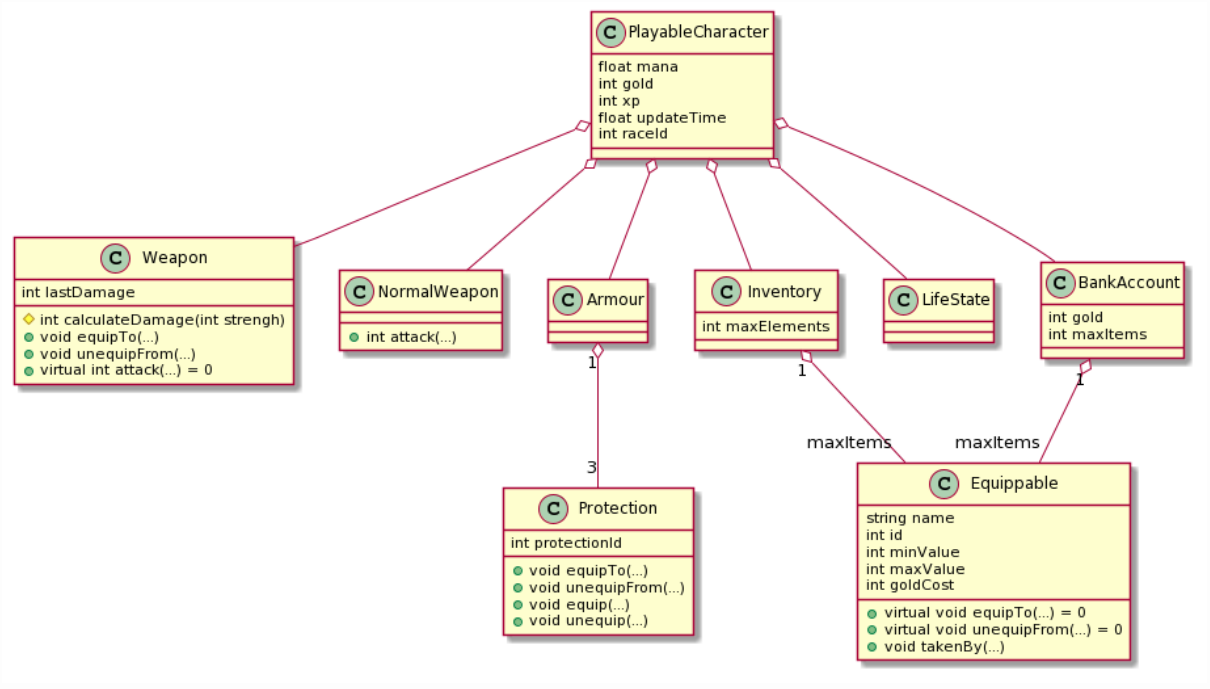
Esta clase representa a los personajes jugables,aquellos personajes controlados por los clientes.Guarda toda la información del jugador (oro,mana,armas,armaduras,etc).

Sus métodos principales son:

- void attack(Character *character) override : Ataca al personaje pasado por parámetro

- void move(Offset& offset) : Se mueve un offset respecto de su posición actual
- void equip(int elementIndex) : Se equipa el item del inventario que se encuentra en la posición elementindex del vector.

Diagrama de clases de la clase PlayableCharacter:



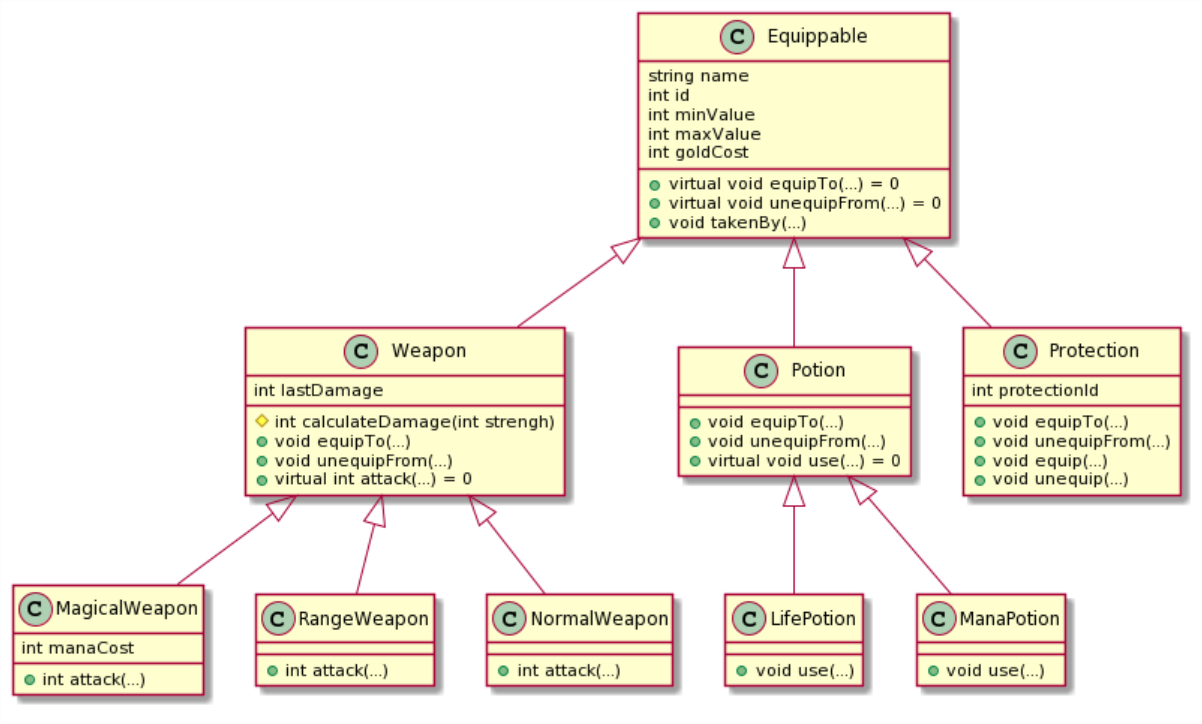
- Equippable:

Esta clase representa todos los items que un jugador puede equiparse: pociones, armaduras,armas,etc.

Sus métodos principales son:

- int randomize() const . Devuelve un valor random entre el valor minimo y el valor maximo del equipable.
- virtual void equipTo(PlayableCharacter *character, int index) = 0.Se utiliza para equipar un equipable a un jugador
- virtual void unequipFrom(PlayableCharacter *character) = 0.Se utiliza para desequipar un equipable de un jugador.

Diagrama de clases de la clase Equippable:



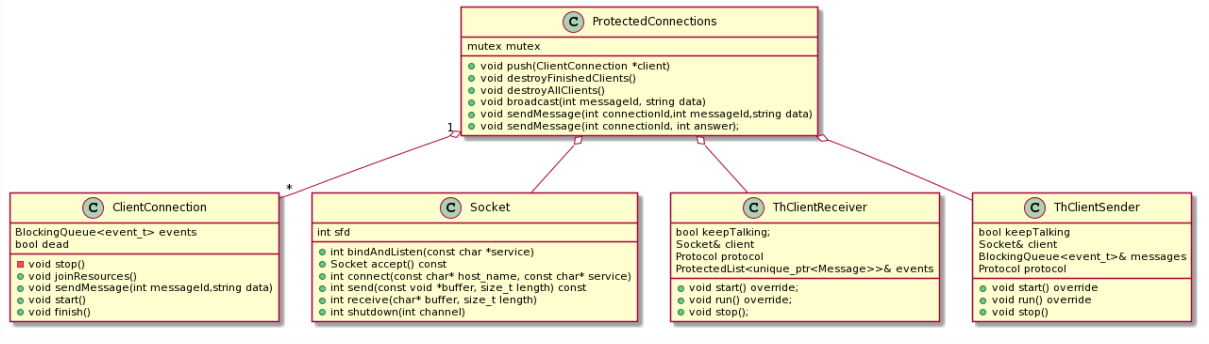
Connection

En este módulo se encuentra todo el código relacionado a la conexión y comunicación del servidor.Las clases mas importantes de este módulo son:

- ProtectedConnections:

  Es un monitor que administra el acceso compartido a las conexiones de los clientes

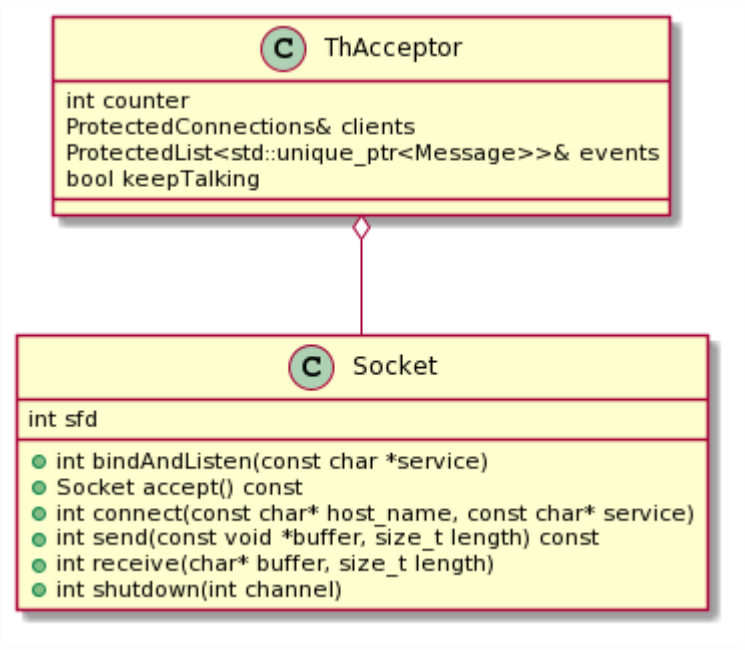
  Diagrama de clases de la clase ProtectedConnections:



- ThAcceptor:

  Es el thread que se encarga de aceptar las conexiones de nuevos clientes y derivarlos al ProtectedConnections.

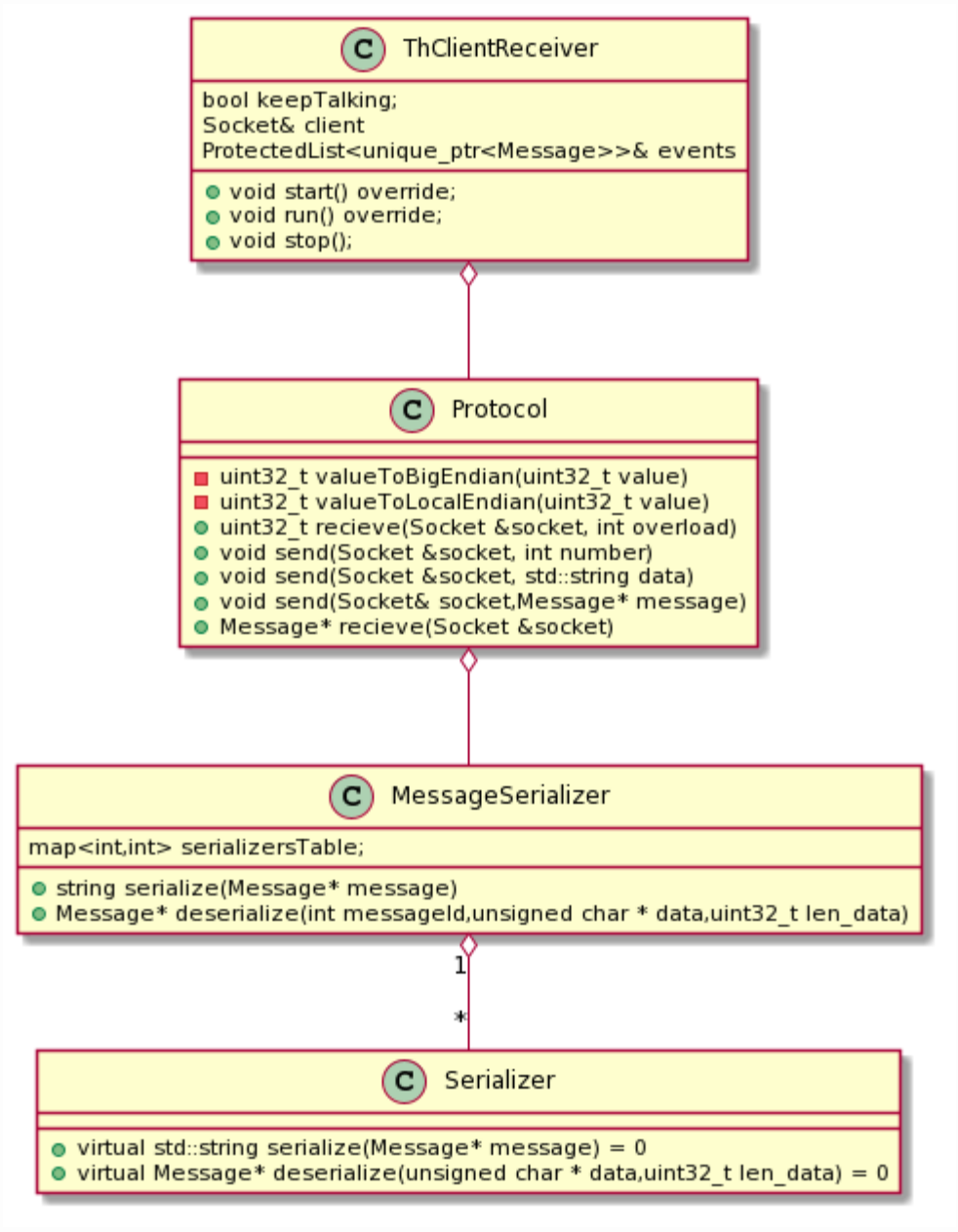
  Diagrama de clases de la clase ThAcceptor:



- ThClientReceiver:

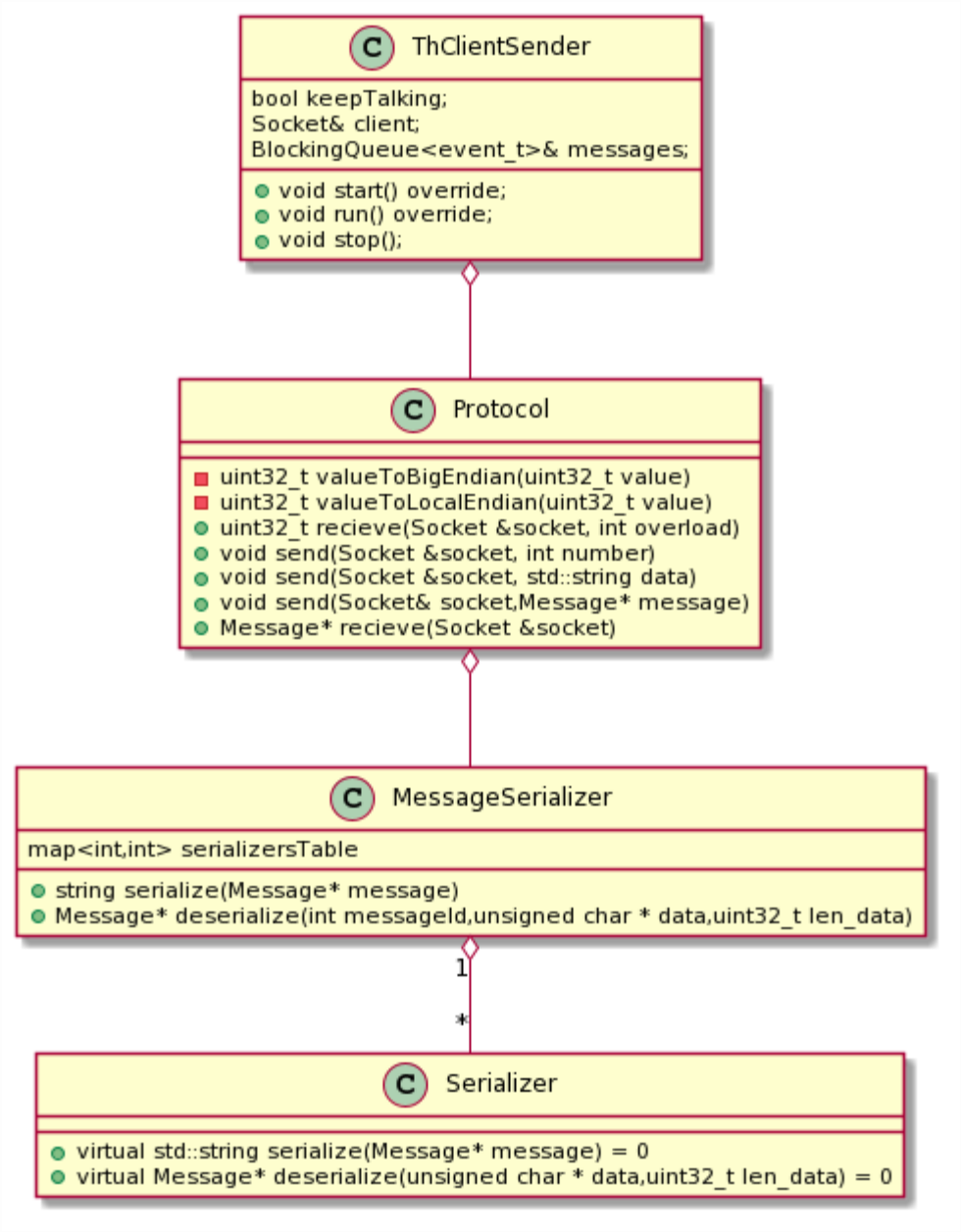
Es el thread que se encarga de recibir los mensajes de su cliente y encolarlos en una lista protegida.

Diagrama de clases de la clase ThClientReceiver:



- ThClientSender:

  Es el thread que se encarga de enviar los mensajes de su cliente asociado.
  Diagrama de clases de la clase ThClientSender:



Common

En este módulo se guardan todos los archivos comunes entre el cliente y el servidor. (thread, socket,blockingQueue,Protocol, etc).
El Protocolo de comunicación entre ellos consiste en: serializar el mensaje utilizando los serializadores que se encuentran en la carpeta serializers de este módulo(cada mensaje tiene un serializador asociado por lo que cada mensaje sabe serializarse correctamente).Luego se envía a través del socket el id del mensaje, y luego se envía los bytes correspondientes a la información del mensaje.