Manual Técnico

Requerimientos de Software

Sistema Operativo

UNIX

Dependencias a instalar

```
sudo apt-get install cmake
sudo apt-get install libsdl2-dev
sudo apt-get install libsdl2-image-dev
sudo apt-get install libsdl2-ttf-dev
sudo apt-get install libsdl2-mixer-dev
sudo apt-get install tiled
```

Comandos útiles - Consola

Instalación

```
git clone https://github.com/franhermani/taller-tp4
bash install.sh
```

Compilación

```
bash make.sh
```

Ejecución del servidor

```
./server ../server/config/config.json
```

Ejecución del cliente

```
./client localhost 8080
```

Diseño

Entidades

Llamamos entidades a todos aquellos objetos que aparecen en el mapa y tienen algún tipo de interacción con el jugador. A continuación, se detalla cada una de ellas:

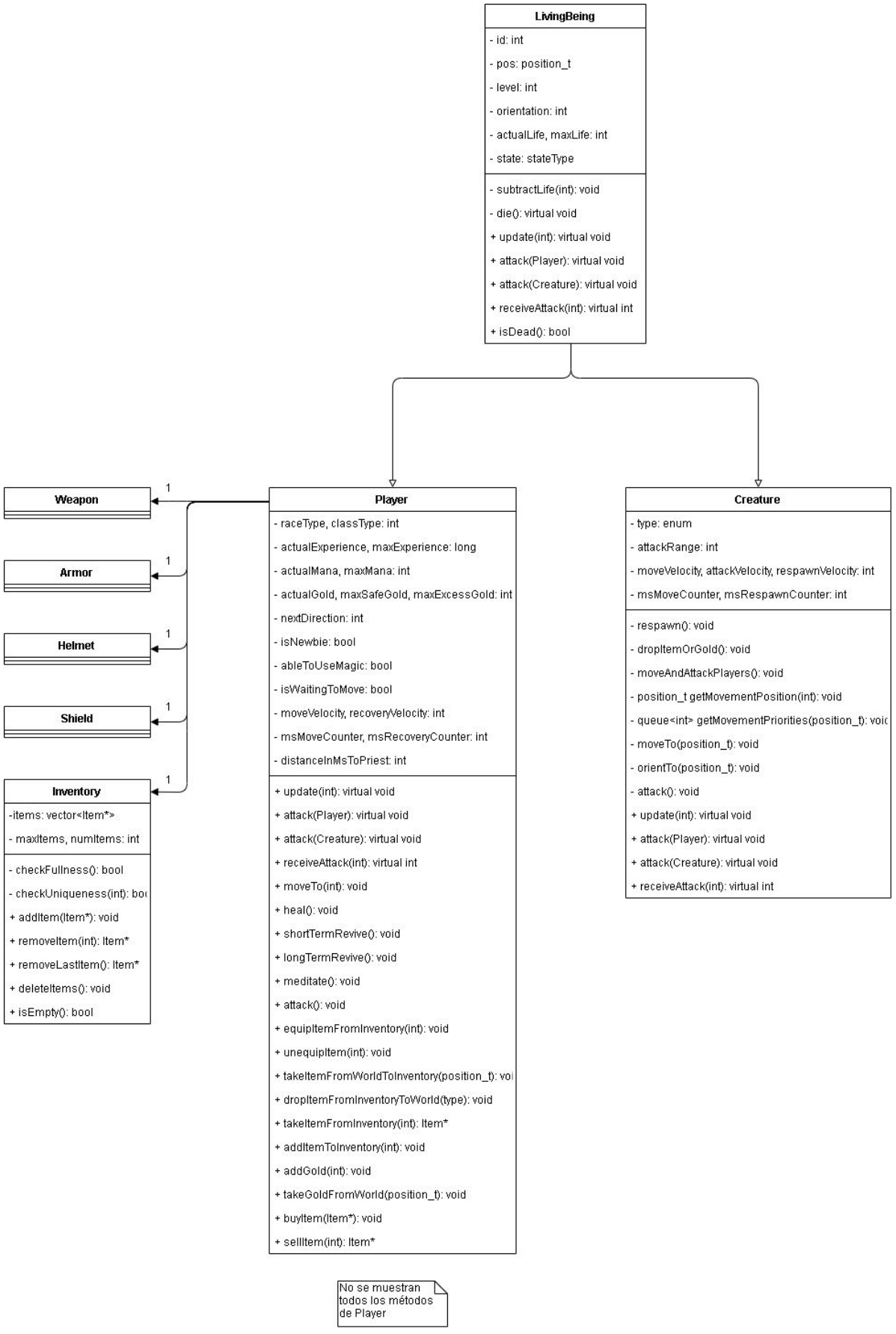
Living Being

Inicialmente, Player y Creature no heredaban de la clase Living Being. Sin embargo, a medida que el código fue creciendo, notamos que compartían mucha lógica de un ser vivo, que puede moverse, atacar y morir.

La motivación final para la creación de esta clase base fue la clase Attack, cuyo owner puede ser tanto un Player como una Creature. De esta forma, pudimos aplicar polimorfismo en World, para ejecutar el comando attack de LivingBeing y delegar su implementación en la clase derivada.

Con respecto a los atributos de velocidad de esta clase, queremos mencionar que están expresados en ms que tarda el objeto en avanzar una unidad. Es decir: ms/unidad. Esta unidad puede ser bloques, puntos de vida, puntos de maná, etc.
Por ejemplo, cada objeto que se mueva a una determinada velocidad, va a tener un contador que se irá incrementando en cada game loop y, cuando llega a su atributo de velocidad, se hace efectivo el movimiento y se reinicia el contador.
Análogamente, para recuperar vida, la unidad será el segundo.

En la siguiente página se puede apreciar un diagrama con todo lo mencionado y más:

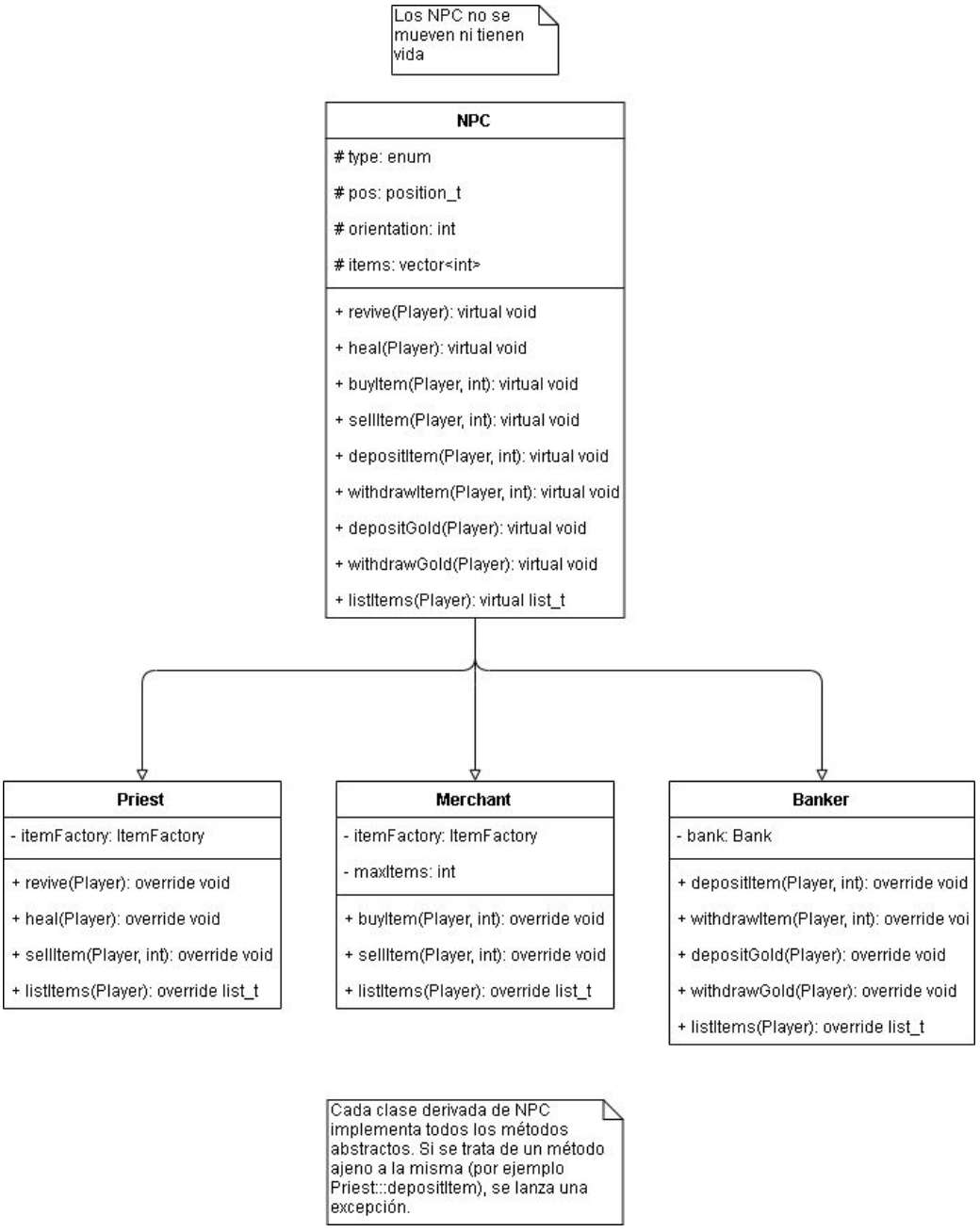

LivingBeing
- id: int
- pos: position_t
- level: int
- orientation: int
- actualLife, maxLife: int
- state: stateType
- subtractLife(int): void
- die(): virtual void
+ update(int): virtual void
+ attack(Player): virtual void
+ attack(Creature): virtual void
+ receiveAttack(int): virtual int
+ isDead(): bool
Weapon
Armor
Helmet
Shield
Inventory
-items: vector<Item*>
- maxItems, numItems: int
- checkFullness(): bool
- checkUniqueness(int): boo
+ addItem(Item*): void
+ removeItem(int): Item*
+ removeLastItem(): Item*
+ deleteItems(): void
+ isEmpty(): bool
1
Player
- raceType, classType: int
- actualExperience, maxExperience: long
- actualMana, maxMana: int
- actualGold, maxSafeGold, maxExcessGold: int
- nextDirection: int
- isNewbie: bool
- ableToUseMagic: bool
- isWaitingToMove: bool
- moveVelocity, recoveryVelocity: int
- msMoveCounter, msRecoveryCounter: int
- distanceInMsToPriest: int
+ update(int): virtual void
+ attack(Player): virtual void
+ attack(Creature): virtual void
+ receiveAttack(int): virtual int
+ moveTo(int): void
+ heal(): void
+ shortTermRevive(): void
+ longTermRevive(): void
+ meditate(): void
+ attack(): void
+ equipItemFromInventory(int): void
+ unequipItem(int): void
+ takeItemFromWorldToInventory(position_t): voi
+ dropItemFromInventoryToWorld(type): void
+ takeItemFromInventory(int): Item*
+ addItemToInventory(int): void
+ addGold(int): void
+ takeGoldFromWorld(position_t): void
+ buyItem(Item*): void
+ sellItem(int): Item*
Creature
- type: enum
- attackRange: int
- moveVelocity, attackVelocity, respawnVelocity: int
- msMoveCounter, msRespawnCounter: int
- respawn(): void
- dropItemOrGold(): void
- moveAndAttackPlayers(): void
- position_t getMovementPosition(int): void
- queue<int> getMovementPriorities(position_t): voic
- moveTo(position_t): void
- orientTo(position_t): void
- attack(): void
+ update(int): virtual void
+ attack(Player): virtual void
+ attack(Creature): virtual void
+ receiveAttack(int): virtual int
No se muestran todos los métodos de Player

NPC

Los NPCs decidimos modelarlos de forma diferenciada a los Players y Creatures ya que no pueden moverse ni morir.

Cabe destacar que la clase base NPC tiene todos los métodos virtuales de sus clases derivadas. Esto fue pensado así para que, en el caso de que un usuario ejecute el comando al NPC equivocado (por ejemplo, Vender dirigido hacia un Banquero), el método lance una excepción y le avise al usuario que esto no es válido.

El siguiente diagrama detalla sus características:



Ítem

Los ítems los modelamos pensando en el equipamiento de un Player. En este sentido es que se realizó la separación en cada uno de los slots posibles que puede equipar un Player.

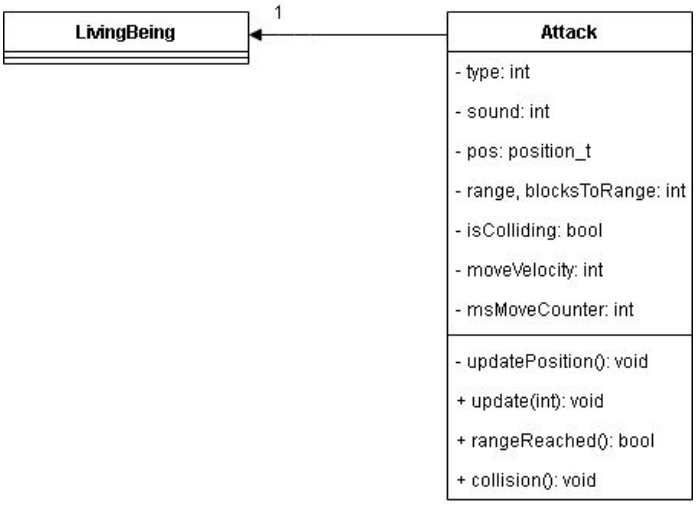
Los distintos tipos de ítems están representados con *enums*, y cada constructor recibe los parámetros de un json para que sea fácilmente configurable.

La clase ItemFactory es quien centraliza la construcción de todos los ítems, y según el tipo recibido le asigna los parámetros apropiados, que luego serán utilizados para las distintas ecuaciones.

Attack

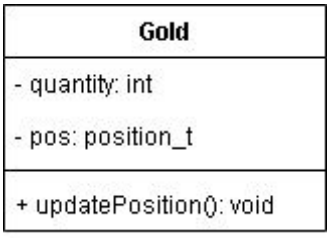
Consideramos importante destacar que tanto los ataques cuerpo a cuerpo como los ataques a distancia están contemplados dentro de la clase Attack. Esto se representa con el atributo *range*.

Luego, a la hora de actualizarlo en el mapa, se hace una diferenciación entre ataques de rango 1 (cuerpo a cuerpo) y ataques de rango mayor a 1 (a distancia), para que la colisión sea más realista.



Gold

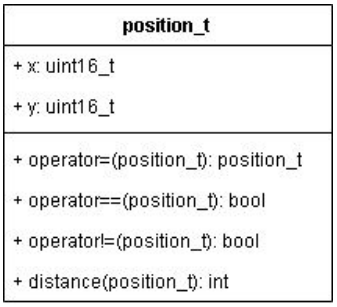
Decidimos diferenciar el oro de los ítems, ya que realmente no tenían casi nada en común, salvo el hecho de que pueden ser recogidos del piso.



Struct position_t

En un proceso de refactorización notamos que siempre la posición X iba de la mano de la posición Y en el mapa. Entonces, para simplificar y encapsular mejor esto, creamos el struct *position_t*.

Todas las entidades cuentan con un atributo de posición en el mapa, el cual está modelado con dicho struct. A continuación, sus detalles:



Mapa

El mapa del juego fue implementado en Tiled. Tomamos de base las texturas del repositorio original de Argentum y agregamos algunos detalles propios.

Hicimos uso de los layers de Tiled para resolver cierta lógica del juego, como las zonas seguras y los terrenos impenetrables.

A continuación, mencionamos todos los layers utilizados:

- *priests*. Posiciones de los sacerdotes en las ciudades.
- *merchants*. Posiciones de los comerciantes en las ciudades.
- *bankers*. Posiciones de los banqueros en las ciudades.
- *unsafe_grounds*. Zonas inseguras donde se puede atacar.
- *safe_grounds*. Zonas seguras (ciudades) donde no se puede atacar.
- *cemetery*. Cementerio a donde respawnean las criaturas luego de morir.
- *impenetrable_grounds*. Terrenos impenetrables que colisionan con entidades.
- *top_layer*. Capa superior para detalles visuales.

Manejo de IDs

Los IDs se representan con un número de 2 bytes. Así, tenemos 65536 disponibles para cada tipo de Living Being:

- Player Del 1 al 65535
- Criatura Del 1 al 65535

Clases y razas

Ambas fueron modeladas con *enums*, ya que tienen una lógica prácticamente nula (sólo la salvedad del guerrero, que no puede meditar ni usar ítems mágicos).

Experiencia

La experiencia es el único número que se almacena en un entero de 4 bytes. Esto es así porque las ecuaciones de experiencia tienen números muy altos y es probable que se llegue a 65535 puntos muy rápido y en un nivel relativamente bajo.

Criaturas

Cuando muere una criatura, no se elimina del mapa. En cambio, pasados unos segundos, se reubica en otra posición (x,y) aleatoria (que no colisione con nada) dentro del cementerio. De esta forma, nos aseguramos que siempre haya la misma cantidad de criaturas en el juego y ahorramos procesamiento y manejo de memoria.

Por otro lado, el nivel de las criaturas está implementado en el GameManager. Éste influye en las ecuaciones pero no en el fair play (solo es entre players).

Características del movimiento de las criaturas:

- Siempre se mueven en dirección al player más cercano
- Si están en su rango de ataque, lo atacan
- No buscan a players fantasmas, ni reviviendo ni dentro de zonas seguras
- Tienen un rango de búsqueda máximo de 50 bloques

Banco

El banco es un recurso compartido por muchos banqueros, con lo cual decidimos protegerlo con un mutex.

Game Exceptions

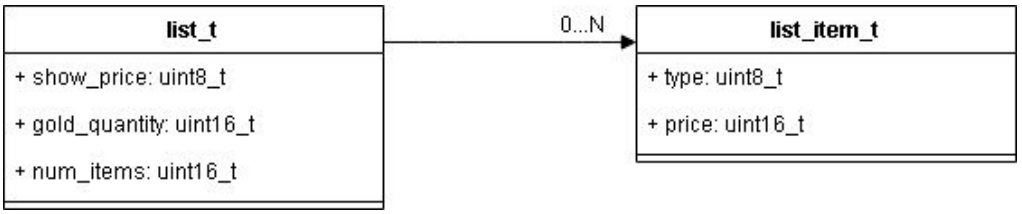
Creamos la clase GameException que recibe un Player ID y un mensaje. El GameManager, luego de ejecutar un comando o durante la actualización del mundo, catchea las posibles excepciones y las encola a una cola de mensajes según el Player ID. Luego, el ClientSender del Player desencola los mensajes y se los envía a su Player (un mensaje por loop, para que sean legibles por el usuario).

Ecuaciones

Las ecuaciones que tenían como parámetro los ms se deprecaron. En cambio, ese paso del tiempo lo realiza la entidad misma (Player o Creature) con un contador, y cuando se llega a la velocidad dada (expresada en ms/unidad), se llama a la ecuación.

Comando Listar

Para enviar la respuesta al comando Listar implementamos la ListException, que funciona de forma análoga a la GameException pero con una estructura en vez de un mensaje. La estructura en cuestión es la siguiente:



Imágenes

Utilizamos *Lazy Initialization* en las imágenes a renderizar. De esta forma, solamente inicializamos las imágenes a medida que las vamos necesitando.
Una vez inicializada, esa imagen creada se reutiliza siempre que se la necesite para renderizar.

Renderizado y sonido

Para renderizado utilizamos la biblioteca SDL2, guardando las imágenes en nuestras implementaciones de superficies dentro del GameImageManager y estampando las necesarias para cada actualización del mundo utilizando el patrón double buffer. Creamos una clase WindowMeasurements para calcular los espacios donde renderizar según la resolución de pantalla del cliente y la cantidad de bloques de visión del jugador.
Para sonidos utilizamos SDL_Mixer.

Idioma

- Todo el código está escrito en inglés, a excepción de los *enums* de los ítems.
- Todo lo relacionado a la interfaz gráfica está en castellano.

Patrones utilizados

Monitor

En el lado del server implementamos un WorldMonitor para que los threads accedan a través de él a la clase World, siempre tomando un lock antes de llamar a cada método.
Luego, las entidades del modelo de dominio (Player, Creature y NPC) acceden directamente a la clase World, ya que no son threads.

En el lado del cliente implementamos un ClientWorldMonitor para que los threads accedan a través de él al modelo local del mapa, siempre tomando un lock antes de llamar a cada método. De esta forma, nos aseguramos que el mapa mostrado al cliente es el correcto.

Command

En el lado del server implementamos el patrón Command al pie de la letra, con una referencia al Player como atributo y un método virtual execute() que recibe una referencia a World como parámetro.

En el lado del cliente implementamos las clases CommandDTO, las cuales carecen del método execute() y, en cambio, se encargan de la serialización en bytes de cada uno de los comandos, según el protocolo propuesto.

Factory

En el lado del server utilizamos este patrón para la creación de ítems (ItemFactory) y para la creación de comandos (CommandFactory).

Game Loop

En el lado del server utilizamos este patrón en la clase GameManager, para simular el paso del tiempo del juego.

De forma análoga, en el lado del cliente lo utilizamos en la clase GameRender, para renderizar las actualizaciones de forma “sincronizada” con el server.

Event Queue

En el lado del server utilizamos este patrón para la recepción y ejecución de comandos por parte de los clientes.
Creamos una cola protegida compartida entre el ClientReceiver de cada cliente y el GameManager.
De esta forma, cada ClientReceiver encola los comandos que va recibiendo, y el GameManager los desencola y ejecuta a medida que van pasando los game loops.

De forma análoga, el cliente comparte una cola protegida entre el GameInputHandler (quien recibe los comandos del usuario) y el ConnectionSender (quien envía los comandos al server).

Double Buffer

Este patrón lo utilizamos en GameRender en el lado del cliente para renderizar el mapa correctamente y que el cambio entre un estado y el siguiente sea imperceptible para el usuario.

Object Pool

Este patrón lo utilizamos en el server para la lógica de las criaturas.

Cuando muere una criatura, no se elimina del mapa. En cambio, pasados unos segundos, se reubica en otra posición (x,y) aleatoria (que no colisione con nada).

De esta forma, nos aseguramos que siempre haya la misma cantidad de criaturas en el juego y ahorramos procesamiento y manejo de memoria.

Threads Cliente

ConnectionSender (único)

Envía la información inicial del jugador para que el server inicialice el juego, y luego envía los comandos ejecutados por el usuario al server.

ConnectionReceiver (único)

Recibe los datos por parte del server para inicializar el modelo local de mundo y luego recibe sus actualizaciones. También recibe mensajes de excepciones del juego.

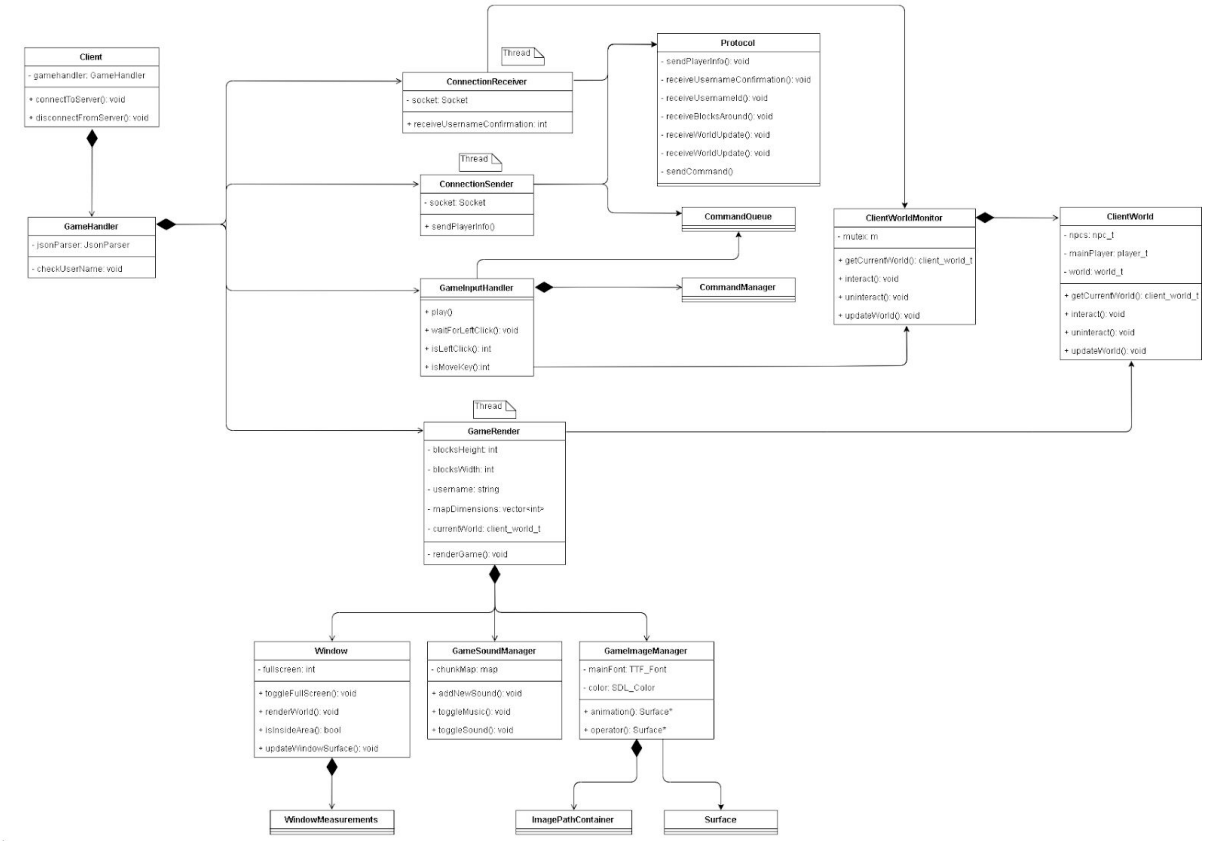
GameRender (único)

Es el encargado de renderizar toda la interfaz gráfica a medida que interactúa con el modelo local de mundo. También provee información de zonas de renderizado.

Es owner de:

- Window
- GameImagesManager
- GameSoundManager

A continuación, mostramos un diagrama con la interacción de todos los threads mencionados:



Threads Servidor

ClientsAcceptor (único)

Acepta conexiones entrantes de clientes y crea los ClientHandler's.

Es el owner de:

- ClientsCleaner
- ClientsBlockingVector (vector bloqueante de ClientHandler's).

Una vez ingresada la 'q' en el servidor, cierra el socket aceptador y llama a ClientsCleaner.stop(), el cual espera a que terminen de jugar todos los clientes conectados. Luego de esto, llama a GameManager.stop() para darle fin a la partida.

ClientsCleaner (único)

Elimina clientes que ya se desconectaron del server.

Tiene una referencia al ClientsBlockingVector de ClientsAcceptor.

ClientHandler (uno por cliente)

Inicializa los threads ClientSender y ClientReceiver y valida el username enviado por el cliente al momento de conectarse. Es quien crea al Player y se lo pasa a GameManager para que lo agregue al mundo.

Tiene una referencia al ClientsBlockingVector de ClientsAcceptor y al GameManager.

ClientSender (uno por cliente)

Envía información relevante al cliente:

- Confirmación del username
- Visión del jugador (en bloques)
- Dimensiones del mapa
- Listado de NPCs
- Actualizaciones del juego al cliente (solo aquello en su rango de visión)
- Mensajes de excepciones
- Respuestas al comando Listar

Tiene un puntero a WorldMonitor, a la cola de excepciones y a la cola de respuestas al comando Listar para enviarle al usuario (todos del GameManager).

ClientReceiver (uno por cliente)

Recibe información relevante del cliente:

- Raza, clase y username
- Comandos ejecutados

Encola los comandos recibidos en la cola de comandos del GameManager.

GameManager (único)

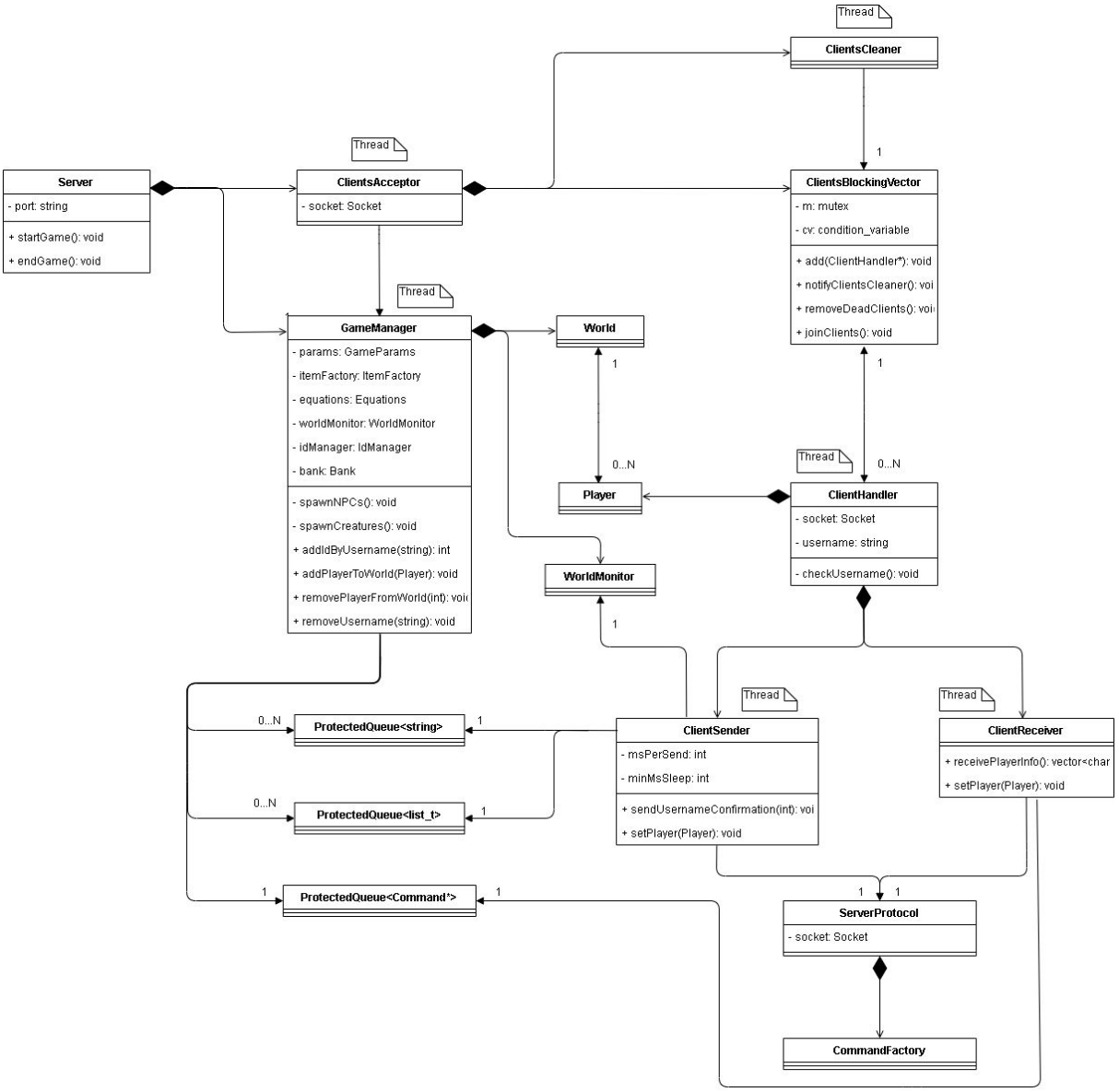
Es quien ejecuta el Game Loop. Realiza varias tareas anexas:

- Crea el mundo, las criaturas y los NPCs.
- Añade los jugadores al mundo
- Ejecuta los comandos recibidos por los clientes que fueron encolados por cada ClientReceiver.
- Actualiza el mundo simulando el paso del tiempo.

Es el owner de:

- GameParams
- ItemFactory
- Equations
- World
- WorldMonitor
- Cola de comandos (única para todos los players)
- Colas de mensajes de excepción (una por player)
- Colas de respuestas al comando Listar (una por player)
- IdManager
- Bank

A continuación, mostramos un diagrama con la interacción de todos los threads mencionados:



Protocolo Cliente - Servidor

Aclaraciones:

- Se utiliza Big Endian para el envío de datos
- El ID del Player no se envía nunca. Esa información la tiene el ClientHandler desde el lado del server, así no es redundante en cada mensaje.

Player info

Cuando se establece la conexión cliente-servidor, el cliente le envía información del player al servidor para que éste lo cree correctamente.

- 1 byte con la longitud del mensaje → Longitud = 1 + 1 + len(username)
- N bytes con el mensaje → Mensaje = raza + clase + username

Comandos

En general, todos los comandos se envían con el siguiente protocolo:

- 1 byte con el enum commandType
- 1 byte con la longitud del mensaje → Longitud = N
- N bytes con el mensaje

Meditar

- Enum: *CMD_MEDITATE*
- Longitud: 0
- Mensaje: <ninguno>

Resucitar

- Enum: *CMD_REVIVE*
- Longitud: 0 ó 2 + 2 = 4
- Mensaje: <ninguno> ó (<posX NPC> + <posY NPC>)

Curar

- Enum: *CMD_HEAL*
- Longitud: 2 + 2 = 4
- Mensaje: <posX NPC> + <posY NPC>

Depositar ítem

- Enum: *CMD_DEPOSIT*
- Longitud: 1 + 2 + 2 = 5
- Mensaje: <enum itemType> + <posX NPC> + <posY NPC>

Retirar ítem

- Enum: *CMD_WITHDRAW*
- Longitud: 1 + 2 + 2 = 5
- Mensaje: <enum itemType> + <posX NPC> + <posY NPC>

Depositar oro

- Enum: *CMD_DEPOSIT_GOLD*
- Longitud: 2 + 2 = 4
- Mensaje: <posX NPC> + <posY NPC>

Retirar oro

- Enum: *CMD_WITHDRAW_GOLD*
- Longitud: 2 + 2 = 4
- Mensaje: <posX NPC> + <posY NPC>

Listar

- Enum: *CMD_LIST*
- Longitud: 2 + 2 = 4
- Mensaje: <posX NPC> + <posY NPC>

Comprar ítem

- Enum: *CMD_BUY*
- Longitud: 1 + 2 + 2 = 5
- Mensaje: <enum itemType> + <posX NPC> + <posY NPC>

Vender ítem

- Enum: *CMD_SELL*
- Longitud: 1 + 2 + 2 = 5
- Mensaje: <enum itemType> + <posX NPC> + <posY NPC>

Tomar ítem u oro

- Enum: *CMD_TAKE*
- Longitud: 1 + 2 + 2 = 5
- Mensaje: <enum takeType> + <posX Item> + <posY Item>

Tirar ítem

- Enum: *CMD_THROW*
- Longitud: 1
- Mensaje: <enum itemType>

Mover

- Enum: *CMD_MOVE*
- Longitud: 1
- Mensaje: <enum moveDirection>

Atacar

- Enum: *CMD_ATTACK*
- Longitud: 0
- Mensaje: ninguno

Equipar ítem

- Enum: *CMD_EQUIP*
- Longitud: 1
- Mensaje: <enum itemType>

Desequipar ítem

- Enum: *CMD_UNEQUIP*
- Longitud: 1
- Mensaje: <enum unequipType>

Protocolo Servidor - Cliente

Aclaraciones:

- Se utiliza Big Endian para el envío de datos
- Todas las estructuras abajo mencionadas están implementadas en el archivo *common/defines/world_structs.h*

Username confirmation

Luego de recibir el username del cliente, el server valida que éste no exista y que haya IDs disponibles. Luego, envía un código de confirmación al cliente:

- 1 byte con el enum usernameCode

Username ID

Si el username era válido, el server envía el ID al cliente para que éste lo almacene y luego pueda identificar a su player.

- 2 bytes con el ID del Player

Visión del player

Luego del ID, el server envía la visión del Player en bloques (alto y ancho).

- 1 byte con la visión en ancho
- 1 byte con la visión en alto

Dimensiones del mapa

El server envía las dimensiones del mapa en bloques (alto y ancho).

- 2 bytes con el ancho del mapa
- 2 bytes con el alto del mapa

Lista de NPCs (npcs_t)

Considerando que los NPCs son estáticos y no tienen vida, los enviamos por única vez al principio del juego y el cliente ya los almacena junto con el mapa.

- 2 bytes con la longitud total del mensaje
- 2 bytes con la cantidad de NPCs
- N bytes con un vector de NPCs

Actualizaciones del mundo (world_t)

Una vez enviada toda la información mencionada en los puntos anteriores, el server envía constantemente actualizaciones del mundo al cliente.

El server le irá enviando a cada cliente las actualizaciones en el mapa, con toda la info necesaria según la posición del Player del cliente en cuestión. Es decir, envía una porción de toda la información del mundo, lo cual hace al envío más eficiente.

Tener en cuenta que el rango de visión (en bloques) de todos los jugadores es el mismo. Luego, según la pantalla que tengan, cada bloque se va a mapear con una cantidad de píxeles distinta, pero siempre nos aseguramos que todos los jugadores tengan el mismo rango de visión. Así no tienen ventajas los que tienen pantallas más grandes sobre aquellos con pantallas más pequeñas.

- 2 bytes con la longitud total del mensaje
- sizeof(player_info_t) bytes con la info particular del Player del cliente
- 2 bytes con la cantidad de Players
- N bytes con un vector de Players (*)
- 2 bytes con la cantidad de Criaturas
- N bytes con un vector de Criaturas
- 2 bytes con la cantidad de Ítems
- N bytes con un vector de Ítems
- 2 bytes con la cantidad de Oros
- N bytes con un vector de Oros
- 2 bytes con la cantidad de Ataques
- N bytes con un vector de Ataques

(*) Incluye al Player del cliente en cuestión.

Es importante destacar que cada vector está formado por estructuras con los atributos adecuados para cada entidad del mundo.

Listado de Ítems (list_t)

Cuando un jugador ejecuta el comando Listar para ver los ítems que tiene para ofrecer un NPC el server le contesta adecuadamente.

- 2 bytes con la longitud total del mensaje
- 1 byte indicando si se muestra o no el precio de los Ítems
- 2 bytes con la cantidad de Oro
- 2 bytes con la cantidad de Ítems
- N bytes con un vector de Ítems

Mensajes del juego

Cuando un jugador intenta realizar una acción no permitida por las reglas del juego, el server lanza una GameException, la catchea y le envía al cliente mensaje apropiado para que éste se lo muestre al usuario.

- 1 byte con la longitud del mensaje
- N bytes con el mensaje de excepción

Ejemplos del protocolo

(Cliente → Servidor) Envío del comando Depositar Ítem

Sea un ítem “Armadura de cuero” y un banquero en la posición (20,30).

Recordando el protocolo:

- 1 byte con el enum commandType
- 1 byte con la longitud del mensaje → Longitud = N
- N bytes con el mensaje

Sabemos que:

- *command_type* = 3 ([1])
- *message_length* = 5 ([2])
- *item_type* = 10 ([3])
- *pos_x* = 20
- *pos_y* = 30

Reescribimos cada parámetro en hexadecimal en Big Endian:

- *command_type* = 03
- *message_length* = 05
- *item_type* = 0A
- *pos_x* = 00 14
- *pos_y* = 00 1E

Escribimos la tira de bytes completa:
03 05 0A 00 14 00 1E

([1]) Ver archivo *common/defines/commands.h*
([2]) *sizeof(item_type) + sizeof(pos_x) + sizeof(pos_y)* = 1 + 2 + 2 = 5
([3]) Ver archivo *common/defines/items.h*

(Servidor → Cliente) Respuesta al comando Listar

Sea el siguiente listado de ítems que ofrece un comerciante:

- Espada
- Hacha
- Martillo
- Armadura de cuero
- Capucha

Sabemos que:

- *message_length* = 20 ([1]) ([2])
- *show_price* = 1 ([2])
- *gold_quantity*= 0 ([2])
- *num_items* = 5 ([2])
- Espada
  - *item_type* = 1 ([2]) ([3])
  - *price* = 50 ([2]) ([4])
- Hacha
  - *item_type* = 2 ([2]) ([3])
  - *price* = 60 ([2]) ([4])
- Martillo
  - *item_type* = 3 ([2]) ([3])
  - *price* = 70 ([2]) ([4])
- Armadura de cuero
  - *item_type* = 10 ([2]) ([3])
  - *price* = 50 ([2]) ([4])
- Capucha
  - *item_type* = 13 ([2]) ([3])
  - *price* = 50 ([2]) ([4])

Recordando el protocolo:

- 2 bytes con la longitud total del mensaje
- 1 byte indicando si se muestra o no el precio de los Ítems
- 2 bytes con la cantidad de Oro
- 2 bytes con la cantidad de Ítems
- N bytes con un vector de Ítems

Reescribimos cada parámetro en hexadecimal en Big Endian:

- *message_length* = 00 14
- *show_price* = 01
- *gold_quantity*= 00 00
- *num_items* = 00 05
- Espada
  - *item_type* = 01
  - *price* = 00 32
- Hacha
  - *item_type* = 02
  - *price* = 00 3C
- Martillo
  - *item_type* = 03
  - *price* = 00 46
- Armadura de cuero
  - *item_type* = 0A
  - *price* = 00 32
- Capucha
  - *item_type* = 0D
  - *price* = 00 32

Escribimos la tira de bytes completa:
00 14 01 00 00 00 05 01 00 32 02 00 3C 03 00 46 0A 00 32 0D 00 32

([1]) *sizeof(show_price) + sizeof(gold_quantity) + sizeof(num_items) + num_items * ((sizeof(type) + sizeof(price))* = 1 + 2 + 2 + 5 * (1 + 2) = 20
([2]) Ver archivo *common/defines/world_structs.h*
([3]) Ver archivo *common/defines/items.h*
([4]) Ver archivo *server/config/config.json*