# 75.42 Taller de programación I
# Documentación técnica


Integrantes:

- Alvarez Windey, Juan - 95242
- Aparicio, Rodrigo - 98967
- Pinto, Tomás - 98757

Ayudante:

Ezequiel Werner

# 1. Descripción General

El proyecto se puede dividir en dos módulos principales que son el **cliente** y el **servidor**. Estos a su vez, pueden dividirse en submódulos.

## 1.1 Cliente

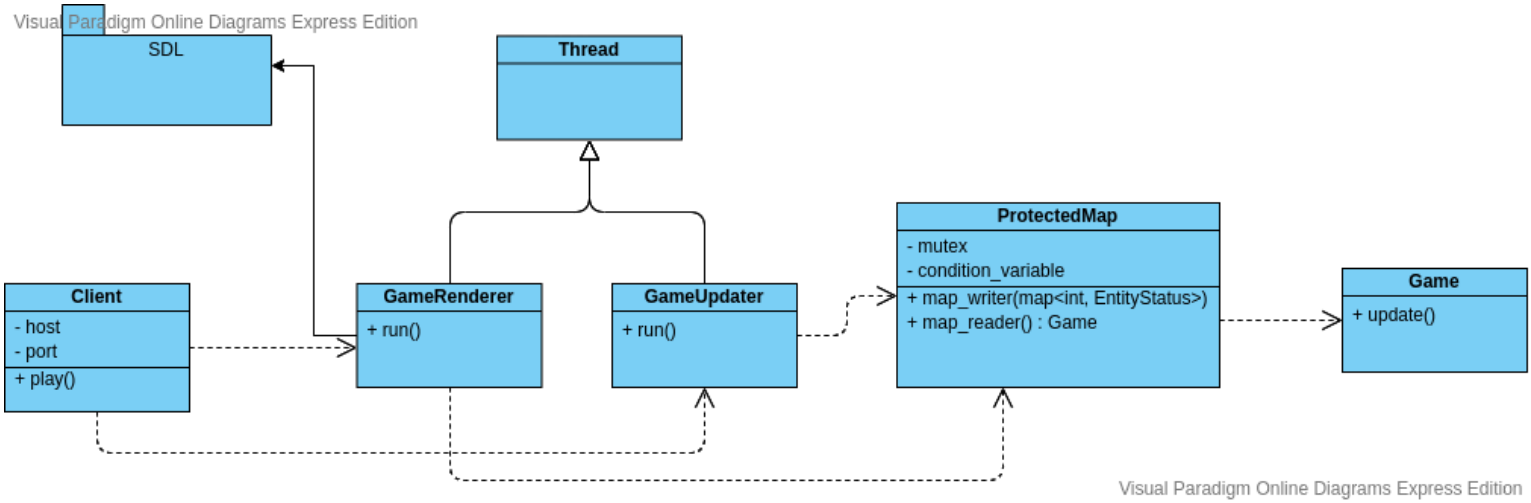
El cliente está representado con la clase Client, donde se corren diferentes threads los cuales realizan las principales tareas del juego como por ejemplo tareas de dibujo, interacción con el usuario y comunicación con el server.
A continuación describiremos cada submódulo con mayor detalle.

### 1.1.1 Módulo de dibujo

En este módulo del cliente se encuentra toda la lógica requerida para poder trasladar lo que ocurre en el juego del lado del servidor a una escena que pueda ser entendida por un usuario que hace uso de la aplicación. En este módulo las clases a destacar son:

- GameRenderer: inicializa e interactúa con la librería elegida para el renderizado del juego (SDL). La instancia de Game a dibujar la copia desde ProtectedMap.
- ProtectedMap: Es un monitor que tiene dos instancias de “Game”, y su propósito es que el GameUpdater pueda escribir la próxima instancia de Game a renderizar y el GameRenderer pueda leer la instancia a dibujar, y que ambos procesos se ejecuten de forma protegida. Este monitor tiene dos instancias ya que se utiliza el patrón “Doble buffer” para la lectura y escritura del estado del juego.
- GameUpdater: recibe las notificaciones del servidor (como por ejemplo los movimientos de los jugadores o npc) y actualiza el mapa con esta información escribiendo sobre ProtectedMap.
- Game: es invocado por el GameRenderer para renderizar toda la información pertinente al juego. Existen dos copias en protected map para asegurar la lectura y escritura segura de las últimas dos instancias del juego .

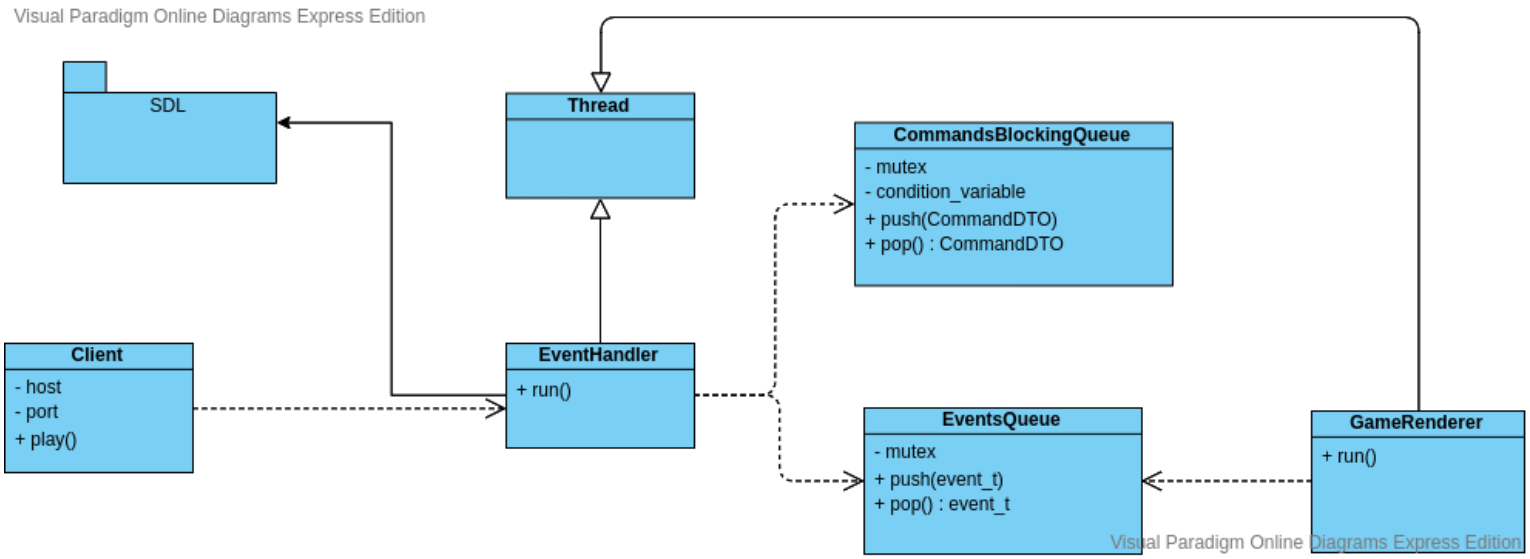
  Tanto GameRenderer como GameUpdater son threads que comienzan a correr desde el método principal (play) de la clase Client.




## 1.1.2 Módulo de interacción con el usuario

Acá podemos mencionar las clases:

- EventHandler: es un thread que se inicializa en el método play del Client cuando comienza el juego y, utilizando la librería SDL, espera los eventos de teclado para determinar si la tecla presionada coincide con alguno de los movimientos o acciones posibles que se pueden realizar en el juego. De ser así se encola un comando al CommandsBlockingQueue, y de ser necesario un evento al EventsQueue.
- CommandsBlockingQueue: es una cola thread safe la cual almacena los comandos que le envía el EventHandler. Los comandos pueden ser un movimiento, un ataque, que se cerró el juego, que el jugador se logueó, etc. Estos comandos se envían al server como se explicará en la próxima sección.
- EventsQueue: es una cola thread safe la cual almacena los eventos que le envía el EventHandler. Esta cola se desencola desde el GameRenderer para determinar qué acción es necesaria realizar. Los eventos pueden ser que el jugador finalizó de jugar o que se seleccionó un item del inventario.




### 1.1.3 Módulo de comunicación Cliente-Servidor

En este módulo se encuentran todas las clases que involucran el traspaso de información de cliente a servidor. En el mismo se destacan las clases:

- Protocol: hace la comunicación entre el cliente y servidor vía socket. Serializa y deserializa los comandos que el cliente envía al servidor.
- CommandsSender: es un thread que se inicializa en el método play del Client cuando comienza el juego. Su función es la de desencolar los comandos de CommandsBlockingQueue y enviarlos al server utilizando el Protocol.

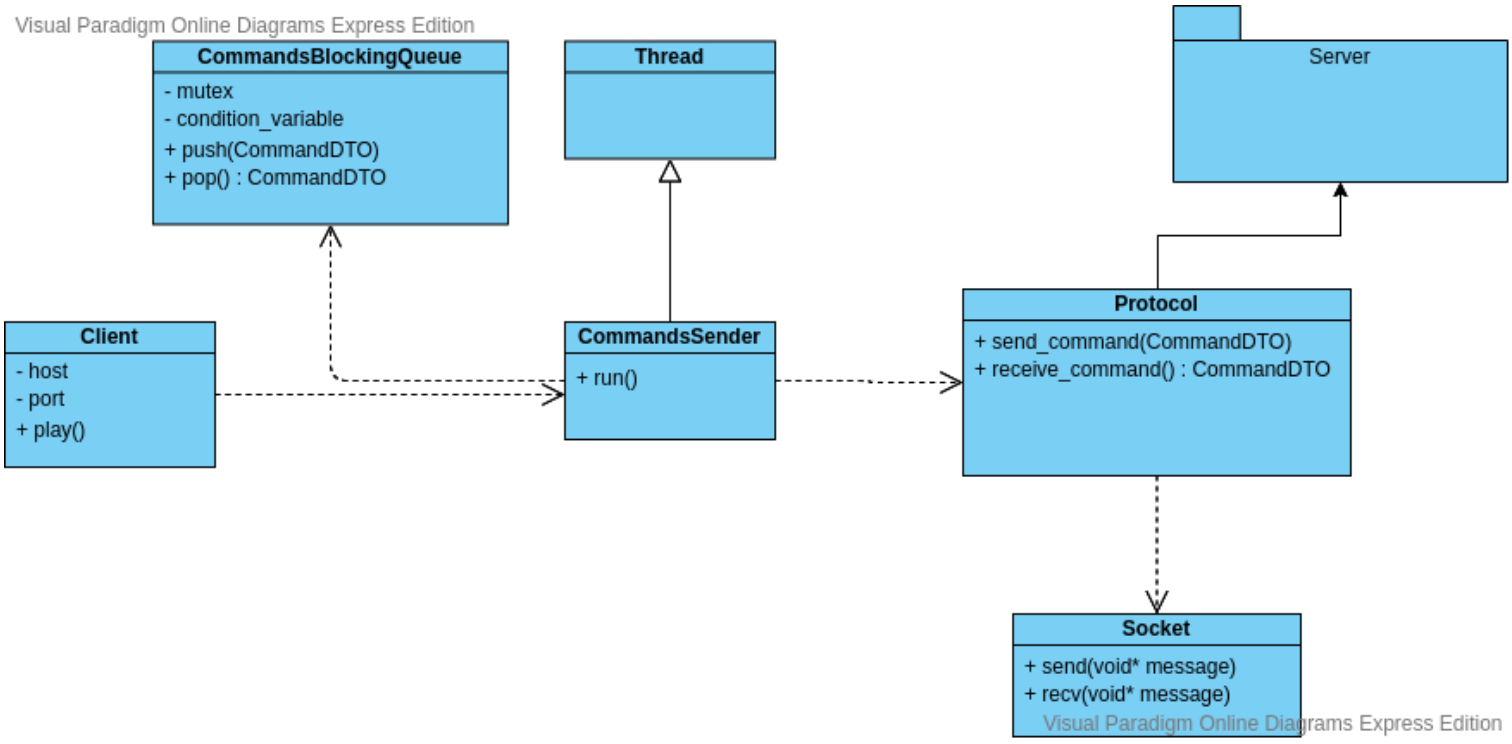
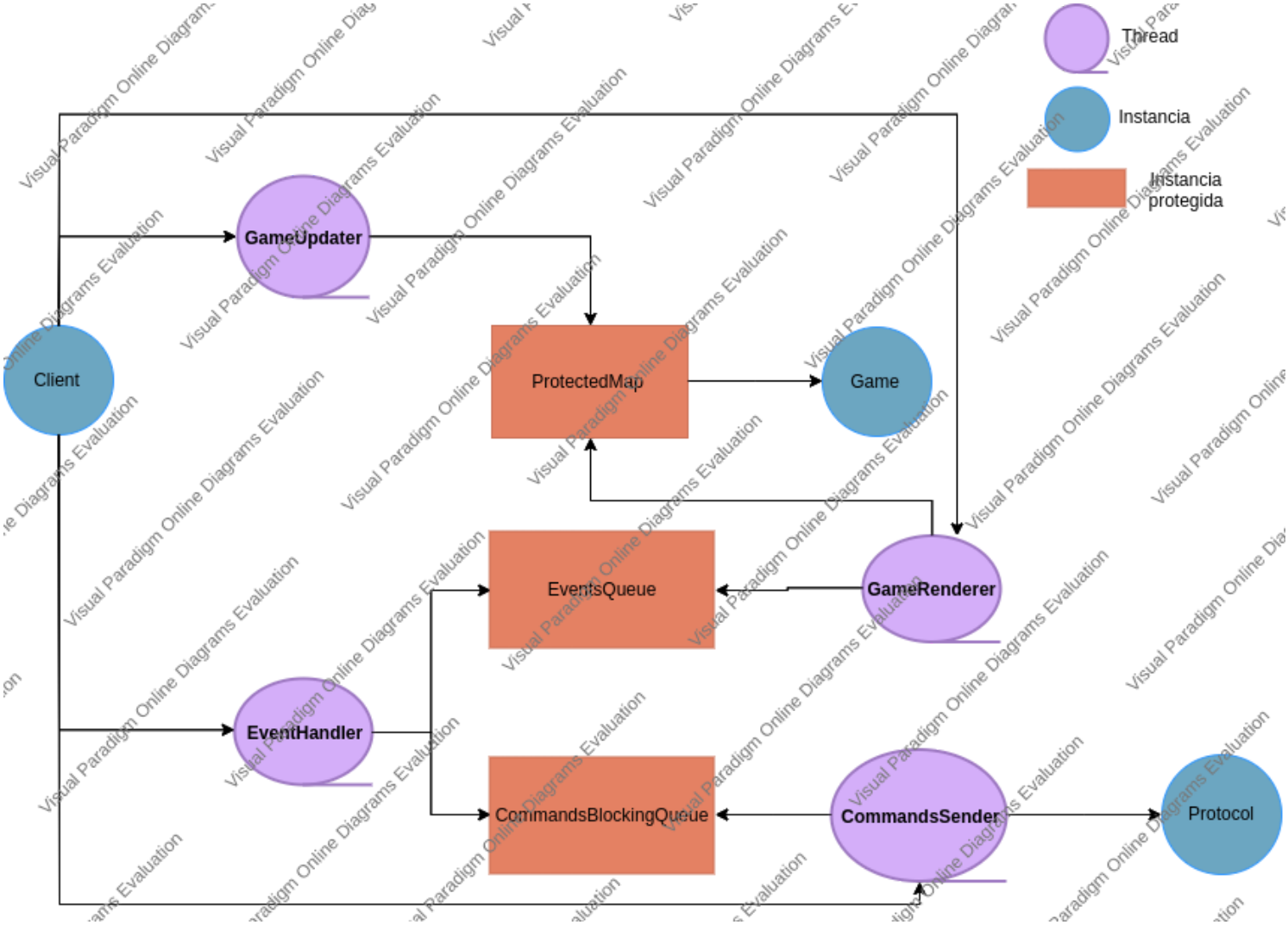
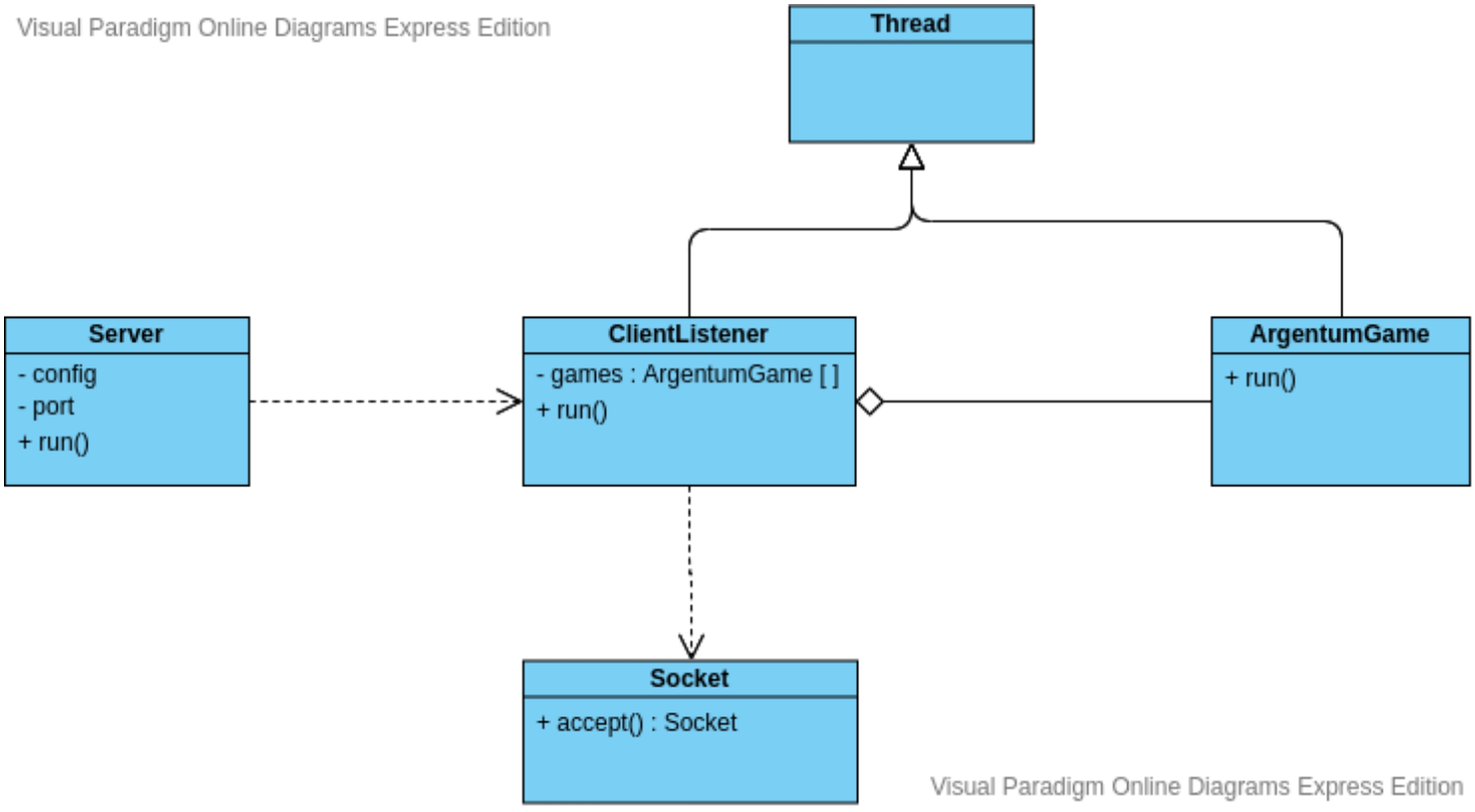

Visual Paradigm Online Diagrams Express Edition
CommandsBlockingQueue
- mutex
- condition_variable
+ push(CommandDTO)
+ pop() : CommandDTO
Thread
Server
Client
- host
- port
+ play()
CommandsSender
+ run()
Protocol
+ send_command(CommandDTO)
+ receive_command() : CommandDTO
Socket
+ send(void* message)
+ recv(void* message)
Visual Paradigm Online Diagrams Express Edition

Diagrama de objeto completo del cliente



## 1.2 Servidor

El servidor inicia con la clase Server la cual comienza a correr la clase ClientListener que es el thread que posee un loop principal en donde en cada iteración espera hasta que un cliente establezca una conexión, siendo este loop bloqueante, ya que se utiliza la aceptación de sockets.
Cuando empieza a correr ClientListener se lanza un thread por cada sala del juego (threads del tipo ArgentumGame). Esta clase almacena el estado (posición, vida, items, etc) de todos los personajes de la sala y permite o no las interacciones propias del juego como matar a alguien, recoger un ítem, etc. Su loop principal fue diseñado siguiendo el patrón de game loop, y para el proceso de comandos de jugadores el command pattern.

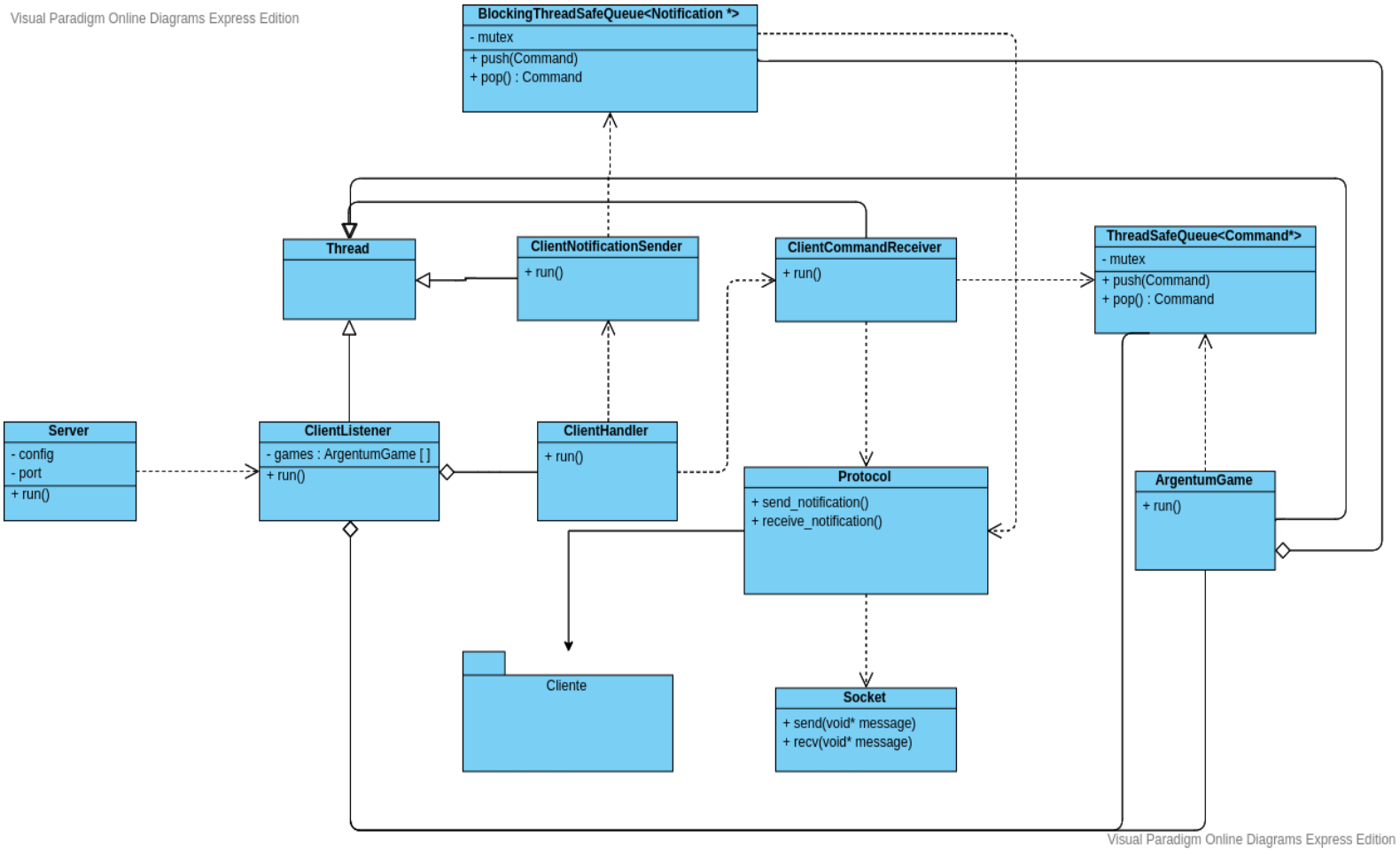


### 1.2.2 Módulo de comunicación con el cliente

Este módulo está representado por la clase ClientHandler. Hay una instancia de ClientHandler por cada cliente. Dentro posee dos threads, uno para enviar notificaciones al cliente y el otro para recibir comandos del cliente. Para la recepción se usa la clase ClientCommandReceiver que recibe pedidos de parte del cliente a través de la comunicación TCP usando el socket que aceptó el módulo anterior, mediante el uso de Protocol. Una vez que los recibe los encola en la clase ThreadSafeQueue<Command*>. Al igual que la clase ArgentumGame, hay una instancia de ThreadSafeQueue<Command*> por cada sala. Desde la instancia de ArgentumGame que se corresponde con la misma sala se desencolan estos comandos y se procesan.

El otro thread es del tipo ClientNotificationSender el cual desencola notificaciones de la clase BlockingThreadSafeQueue<Notification *> (que también hay una por cada cliente) las cuales fueron encoladas por ArgentumGame. Una vez desencolada la notificación se la envía al cliente usando el Protocol, quien deserializa y serializa las notificaciones para que se puedan interpretar desde el cliente.

En un principio ClientHandler era un thread pero decidimos separarlo en dos threads (el Receiver y el Sender) ya que queríamos hacer un envío y recepción vía socket al mismo tiempo.



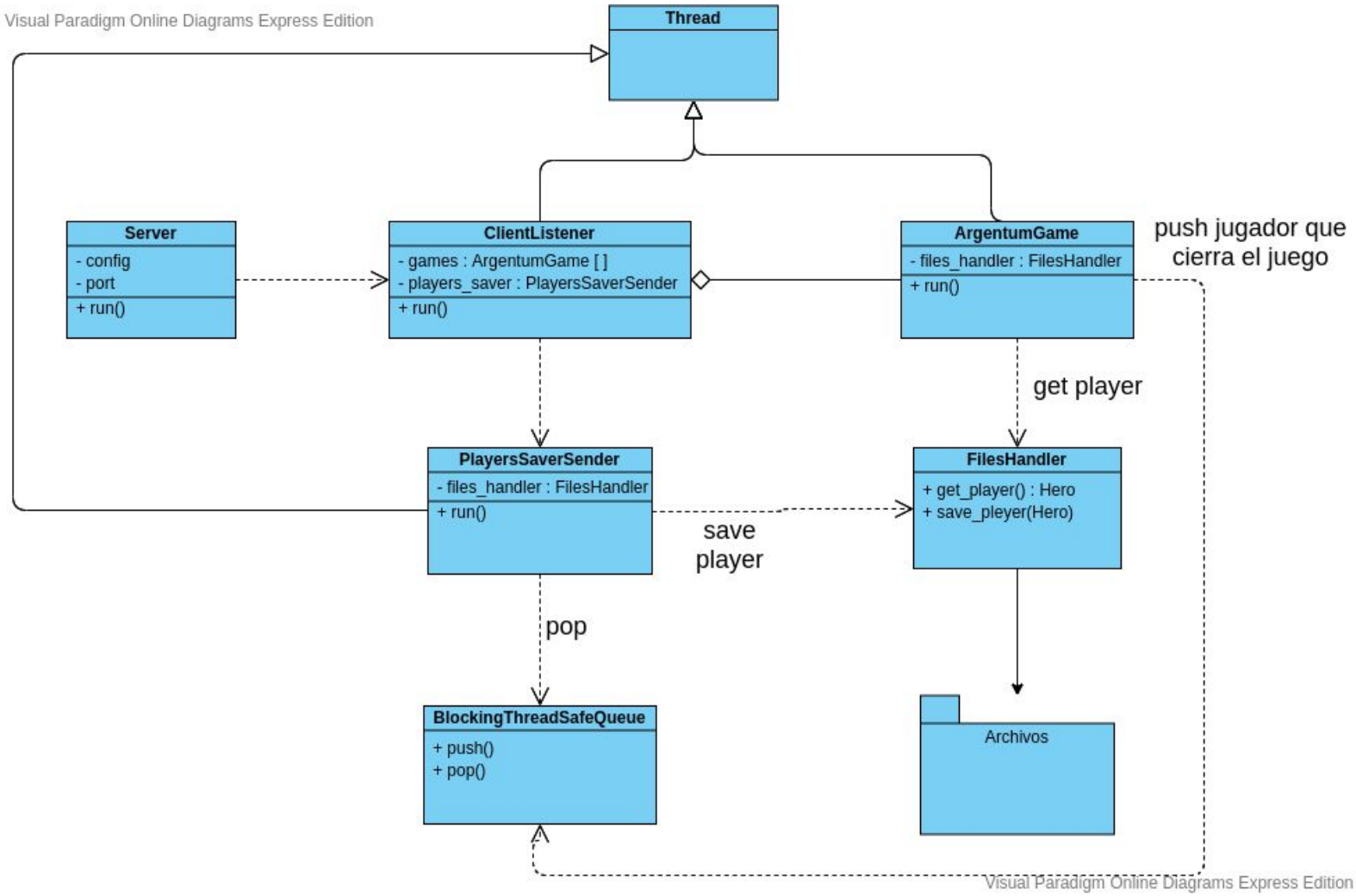

## 1.2.3 Módulo de persistencia

El servidor almacena la información de los jugadores para que cuando regresen puedan continuar desde donde estaban. Para lograr esto tenemos dos archivos, uno donde guardamos la información de todos los jugadores y el otro archivo es un mapa donde se guarda la posición donde está almacenada la información de cada jugador en el archivo de jugadores.

En primer lugar tenemos la clase FilesHandler quien interactúa con los archivos en disco. A este archivo se le pide tanto obtener un jugador como guardarlo. Ambos métodos están protegidos mediante un mutex para evitar la escritura del archivo desde más de un hilo al mismo tiempo ni para leerlo mientras se está escribiendo.

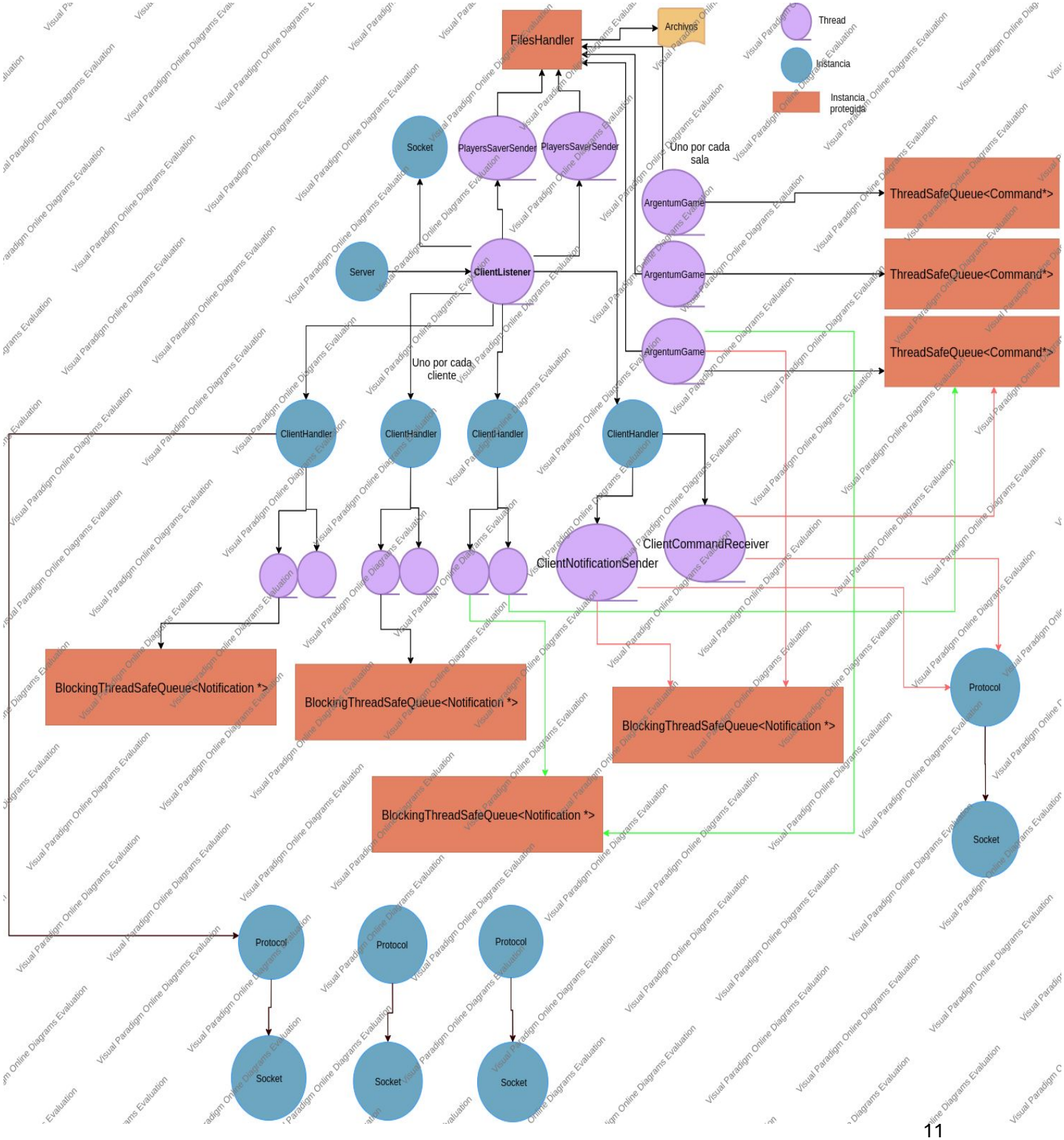
Quienes interactúan con FilesHandler son dos clases: ArgentumGame y

PlayersSaverSender. Cuando un usuario inicia sesión, la clase ArgentumGame le consulta a FilesHandler si el usuario ya existe para evitar crear uno nuevo. Por otra parte, PlayersSaverSender es quien envía los jugadores para que sean almacenados. Este último es un thread que se levanta cuando comienza el juego, dentro de ClientListener. Está dormido hasta que un jugador sale de juego y entonces los datos de este encolan en una cola bloqueante para que PlayersSaverSender lo desencole y lo guarde.



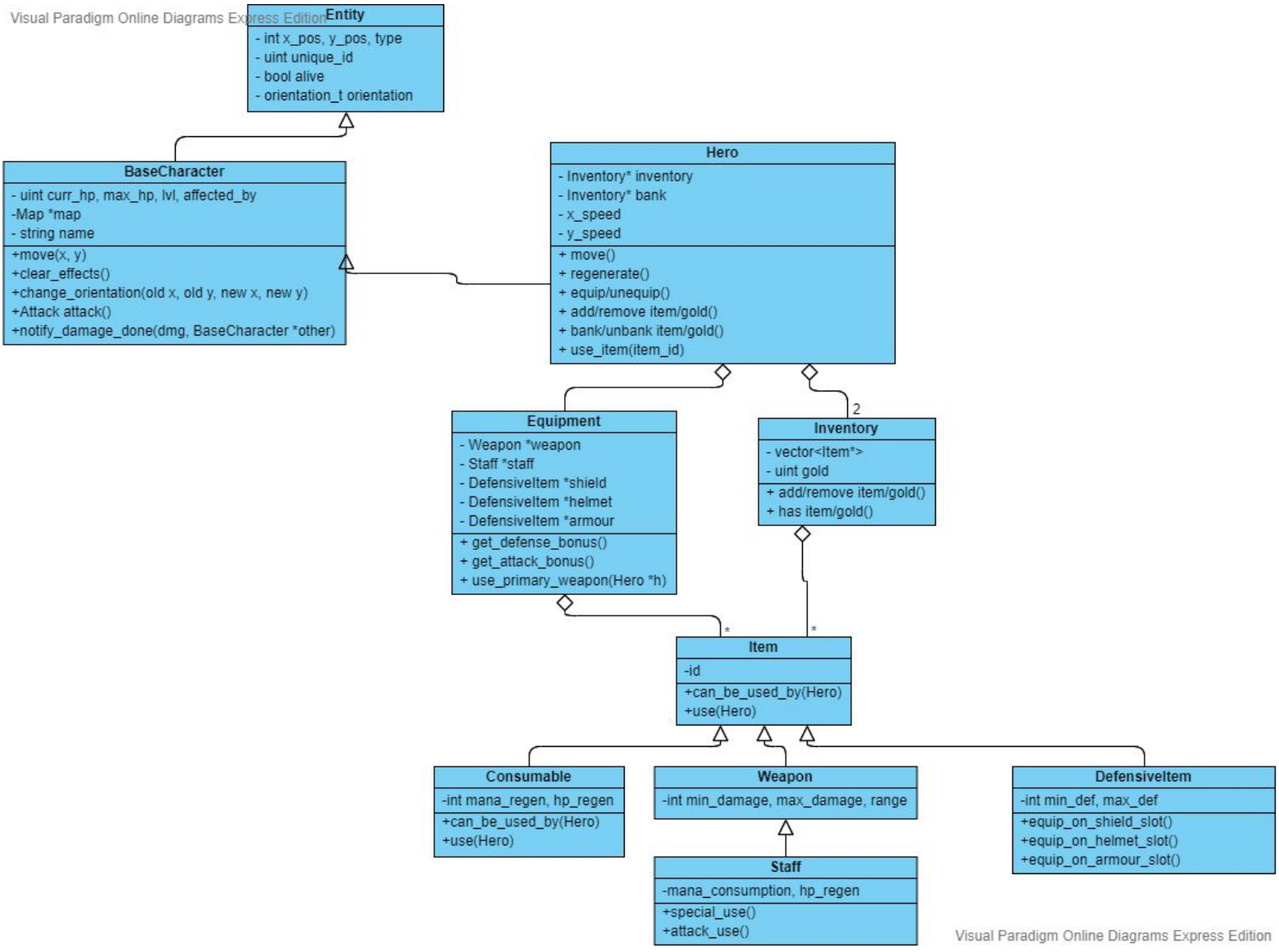

Hay otro feature que consiste en salvados periódicos a todos los jugadores. Para realizar esto, ClientListener levanta otro thread también del tipo PlayersSaverSender pero en lugar de estar esperando jugadores en la cola, duerme por n tiempo y cuando se despierta recorre a todos los jugadores activos y se comunica con el FilesHandler para que este mismo los almacene en disco.

# Diagrama de objeto completo del servidor



# 2. Modelo de negocio

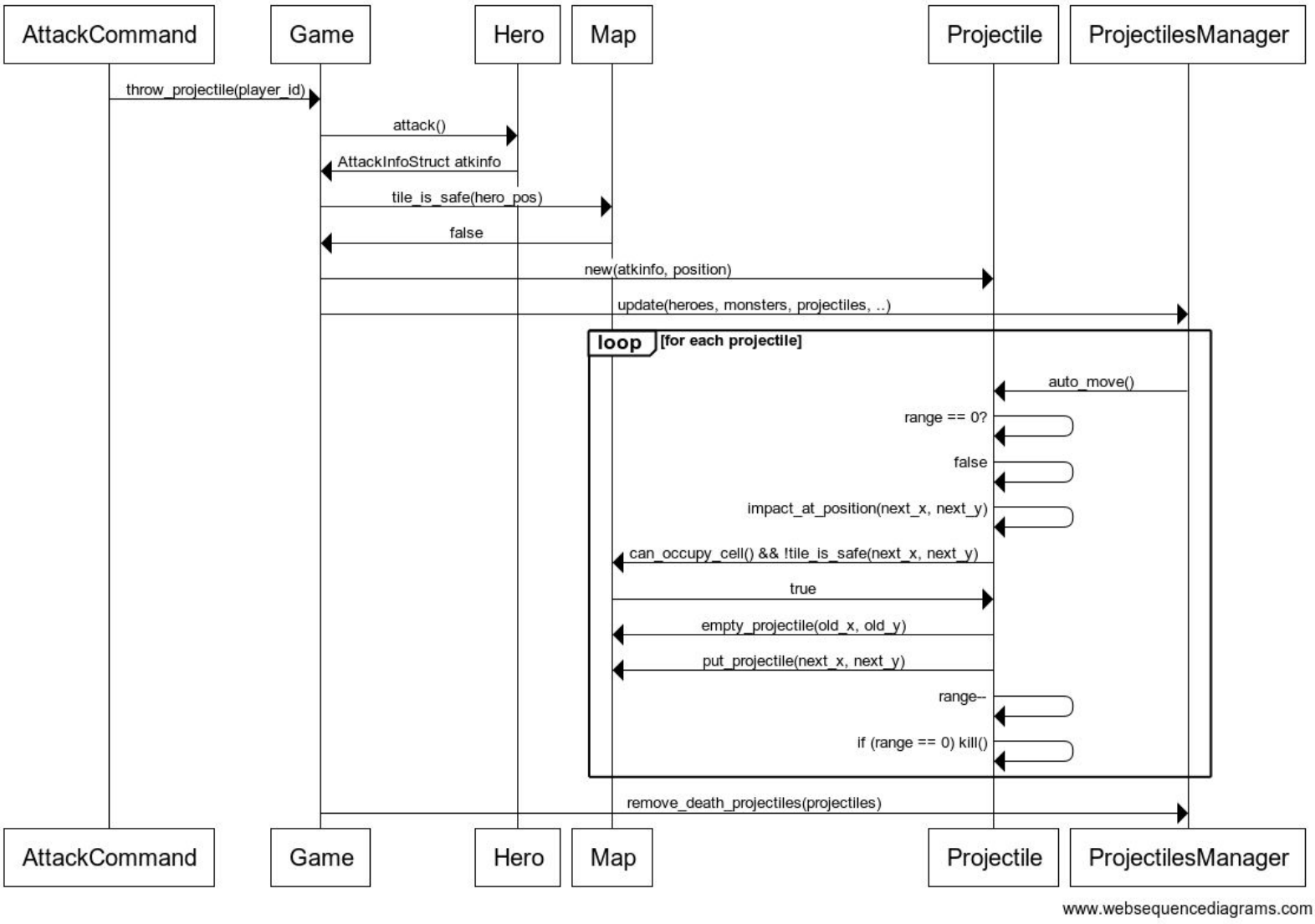
A continuación se encuentra el diagrama de clases de las principales entidades de negocio del juego. Por simplicidad se omitieron algunas subclases que también utilizamos, como por ejemplo la clase Monster que hereda de BaseCharacter.



También vamos a presentar unos diagramas de flujo con interacciones que se producen en el servidor.

Cuando un héroe lanza un ataque, se crea un proyectil cuya estado es manejado por la clase ProjectilesManager en cada loop del juego:

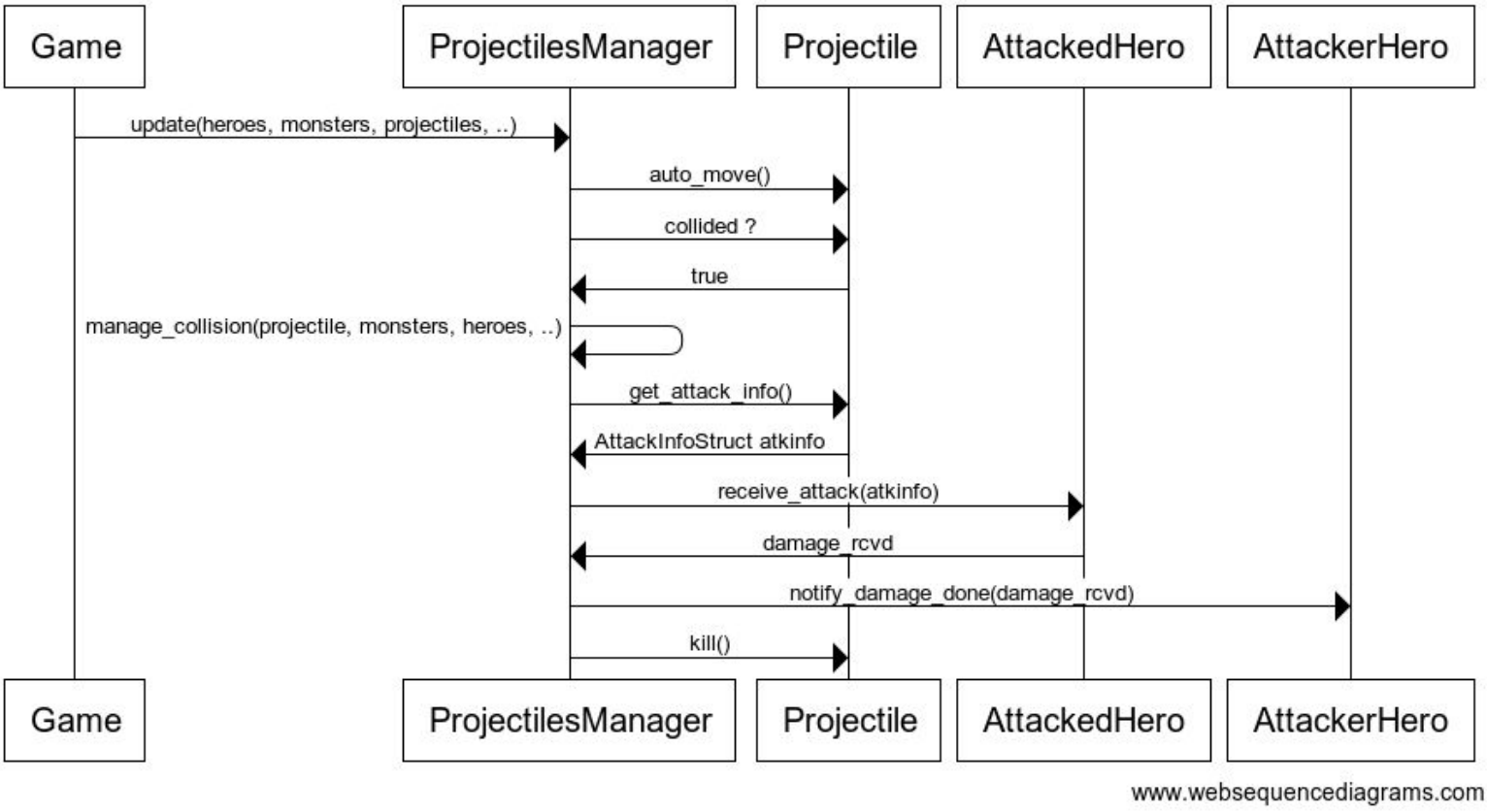
**Diagrama de flujo (Vida de un proyectil)**




www.websequencediagrams.com

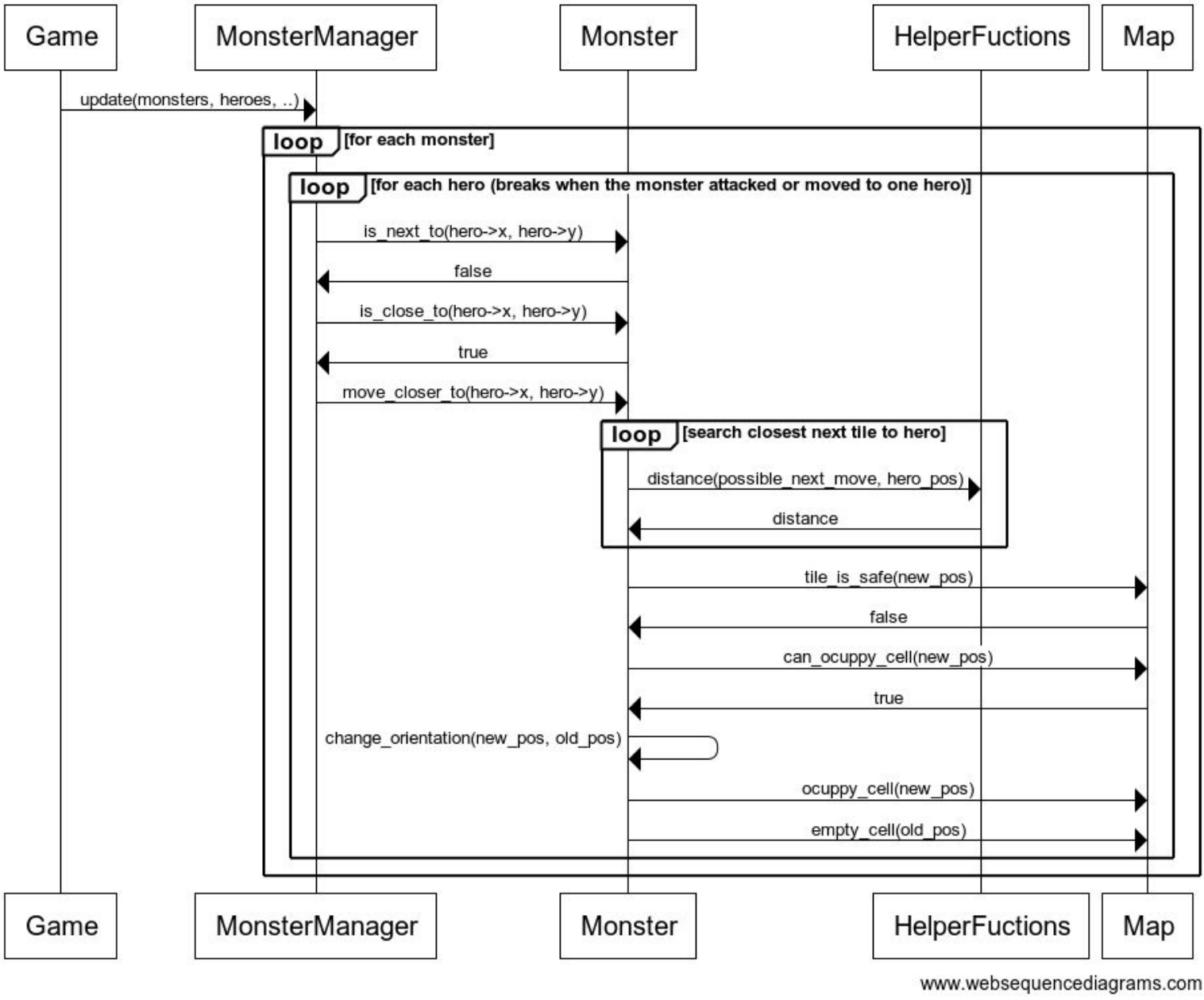
A continuación vemos qué sucede cuando un héroe dispara un proyector y este impacta a otro héroe:

**Diagrama de flujo (Proyectil de un heroe impacta otro heroe)**



Algo importante que hace el server es manipular a todos los monstruos. Realizamos un diagrama de flujo donde se muestra el algoritmo de cómo se hace para que los monstruos se acerquen a los héroes.

**Diagrama de flujo (Acercamiento de un monstruo a un heroe)**



# 3. Flujo de comunicación vía protocolo

Observamos la lógica utilizada para manejar el envío y recepción de comandos utilizando el protocolo: