Universidad ORT Uruguay

Facultad de Ingeniería

Ingeniería en Sistemas

Programación de Redes

Obligatorio 1

Marcel Cohen - 212426
Facundo Arancet - 219606

Grupo M6A
Docente: Roberto Assandri

Índice

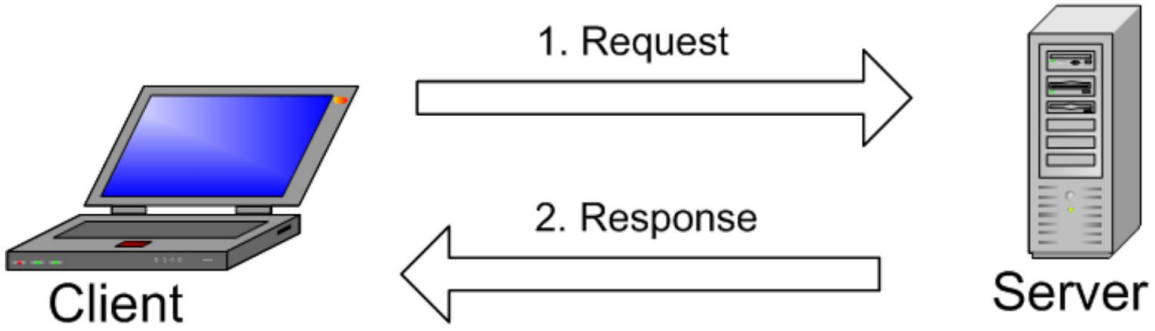
Introducción

El sistema consiste de dos aplicaciones, un cliente y un servidor.

El servidor es el que contiene el juego y toda su lógica. Mantiene el registro de los jugadores que se dieron de alta y administra las partida.
Los clientes permiten al usuario conectarse al servidor, mediante este, los usuarios envían peticiones para jugar el juego y acceder a otras funcionalidades que el servidor brinda.
El servidor procesa las solicitudes y devuelve mensajes de respuesta, el cliente de acuerdo a esta le notifica al usuario.
Las dos aplicaciones funcionan por consola.

Arquitectura

La arquitectura básica se puede representar mediante el siguiente diagrama.



Se trata de una arquitectura básica en la cual un servidor y muchos clientes se conectan utilizando sockets que se comunican mediante el protocolo TCP.

Se definió un protocolo para definir la forma en que se comunican las dos partes, básicamente, para que el servidor pueda entender que le solicita el cliente y pueda enviar una respuesta acorde, la cual el cliente tambien sabra procesar. De este protocolo se hablará en detalle en otro apartado más adelante.

También, se utilizó una abstracción, la interfaz IConnection, que representa la conexión subyacente entre las aplicaciones, que en esta versión es TCP/IP. Pero este tipo de conexión podría cambiar, ya que es un detalle de implementación. Es por esto, que la abstracción utilizada permite que la lógica del negocio no conozca los detalles de la conexión, de esta forma se logra un mejor diseño que favorece la mantenibilidad.

Protocolo

Según lo solicitado en la letra, se especificó un protocolo propietario orientado a caracteres implementado sobre TCP/IP. Este es el “acuerdo” que cumplen las aplicaciones cliente y servidor para establecer la comunicación.

Justificación

Se utilizó el protocolo recomendado en la letra y no se modificó, contrario a la sugerencia de la letra misma. Se optó por mantener el protocolo original porque resultó adecuado. Si bien la sección del tipo de encabezado (RES o REQ) podía sustituirse por un binario, hace que el protocolo sea más fácil de entender y facilita el debug. Como se tienen más de 10 comandos, pero se sabe que nunca habrán más de 99, se conservaron los 2 caracteres para representar el tipo de comando.
Por último, también se conservaron los 4 bytes para indicar el largo del dato a enviar, esto especifica que el largo máximo de los datos es de 9999 bytes. Se puede discutir que es un desperdicio de bytes ya que la mayoría de los comandos, como los movimientos en la partida y el enviado de nicknames para registrar e ingresar no llevan datos, o llevan datos pequeños. Por otro lado, también existen mensajes con datos muy largos, como fotos, o listas de jugadores jugando o registrados en el sistema.
Teniendo un largo máximo de dato pequeño implicaría enviar una gran cantidad de paquetes con fragmentos del dato, lo cual se creyó que puede resultar en la saturación del medio, y esto bajaría la performance. A su vez, un largo máximo que soportará enviar todo tipos de datos en un solo paquete sería un desperdicio de bytes para los mensajes que envían datos pequeños, que son la mayoría.

En conclusión, para la definición del protocolo, se trabajó bajo la duda, por eso, el equipo no se enfocó en encontrar el protocolo ideal, sino en abstraer el protocolo, y desarrollar la aplicación, de forma que adaptar el protocolo sea sencillo. Para lograr esta abstracción se definieron clases como Package y Header, en el paquete Protocol, ahí se encuentran todas las constantes del sistema que determinan la configuración del protocolo.

Descripción del protocolo

El protocolo consta del uso de paquetes, donde un paquete (Package) se constituye por un cabezal (Header) y los datos (Data). Un cabezal tiene un fijo de 9 bytes (o caracteres si es codificado en UTF-8), mientras los datos pueden tener un largo variable.

	Cabezal			**Datos**
Nombre	Type	Command	Length	Data

Valores	REQ/RES	00-99	0000-9999	Variable
Largo	3	2	4	Variable

Comando	**Operación**	**Propósito**
00	AUTHENTICATE	Solicitar autenticación, lleva como dato el nickname con el que se ingresara.
01	ENTER_OR_CREATE_ MATCH	Solicita entrar a una partida activa o crearla e ingresar.
02	ADD_USER	Solicita ingresar un usuario, lleva como dato el nickname del usuario a registrar.
03	LOG_OUT	Solicita desconectarse del servidor.
04	ERROR	Informa sobre error.
05	CHOOSE_MONSTER	Solicita jugar como monstruo.
06	CHOOSE_SURVIVOR	Solicita jugar como sobreviviente.
07	RETURN_TO_MENU	Solicita volver al menú principal.
08	PLAYER_ACTION	Solicita ejecutar una acción sobre un jugador.
09	OK	Mensaje que indica que la solicitud se procesó correctamente.
10	REGISTERED_USERS	Solicita la lista de los usuarios registrados en el sistema.
11	IMG_JPG	Indica que el dato a enviar corresponde a una imagen.
12	END_MATCH	Informa el fin de una partida.
13	NOTIFICATION	Indica que el paquete contiene un mensaje de notificación.
14	IN_GAME_PLAYERS	Solicita la lista de los jugadores en partida.

Manejo de concurrencia

Manual de uso y reglas del juego

Descripción del juego

En este apartado, se describen todas las reglas de la lógica de negocio. Se incluyen las suposiciones del equipo.

El juego es un MUD (Multi User Dungeon) sencillo. En el mismo existen dos roles que combaten entre sí, “Survivors” y “Monsters”. El objetivo de un jugador cuyo rol es monster es eliminar a todos los jugadores del mapa, mientras que el objetivo de un survivor es sobrevivir hasta que el tiempo se termine o hasta que ya no hayan jugadores monster en el mapa.

El juego se desarrolla en partidas de tiempo fijo. Si al terminar el tiempo, no existen survivors y más de un monster, entonces el juego finaliza en empate.

Cuando un jugador ingresa a la partida, es colocado en una posición al azar del mapa. Durante un período de tiempo fijo, el jugador puede desplazarse por el mapa y ver a los otros jugadores, pero no puede atacarlos. En este período es en el que nuevos jugadores pueden ingresar a la partida. El juego verifica si se puede empezar una partida nueva con los jugadores actuales en el mapa cada un periodo de tiempo. Si no se puede comenzar una partida nueva, el cronómetro se reinicia.

Una vez que la cantidad de jugadores es la adecuada para comenzar la partida (más de un monster, o al menos un monster y un survivor), se le informa a los jugadores que ya tienen disponible la acción de atacar. A partir de este momento, el servidor ya no permite ingresar nuevos jugadores al mapa.

Los jugadores tienen puntos de vida, se pueden desplazar por el mapa y también atacar. El ataque de los jugadores se realiza en todos los lugares en torno a la posición del jugador (las 8 posiciones próximas al jugador).
La acción de ataque de un survivor solo daña monsters (no friendly fire), mientras que la acción de ataque de un monster daña a survivors y monsters.

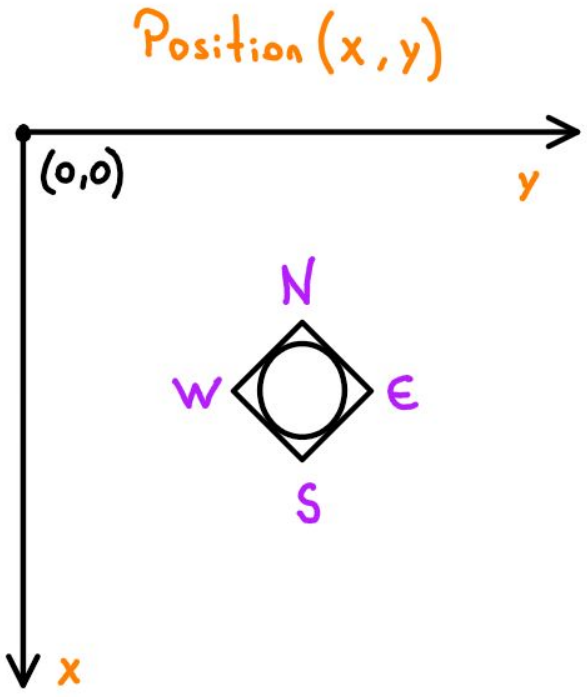
Un jugador muere cuando sus puntos de vida llegan a 0. Un jugador que muere, pierde y es expulsado de la partida.
Los puntos de vida iniciales de un survivor son 20 y sus puntos de ataque son 5.
Los puntos de vida iniciales de un monster son 100 y sus puntos de ataque son 10.

Acciones de un jugador

El movimiento de un jugador es efectuado utilizando un Tank Controller. Por lo tanto, tiene una dirección en la que apunta y las opciones de moverse hacia adelante o hacia atrás.
Los comandos para controlar las acciones de un jugador son:

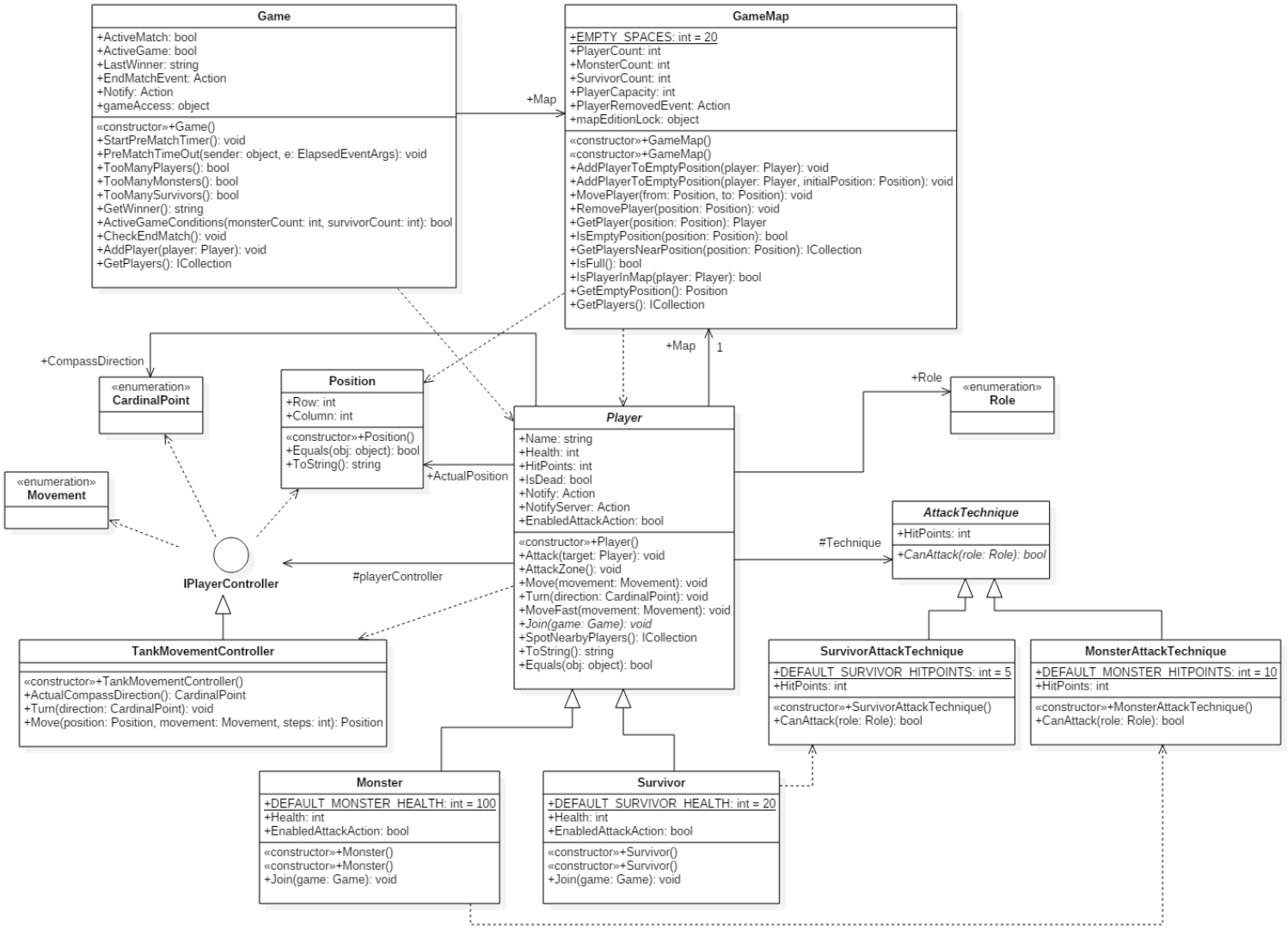
Acción	*Comando por consola*
Moverse hacia adelante	f
Moverse hacia atrás	b
Moverse rápido hacia adelante	ff
Moverse rápido hacia atrás	bb
Mirar hacia el norte	w
Mirar hacia el este	d
Mirar hacia el oeste	a
Mirar hacia el sur	s
Atacar	e

Mapa del juego



El juego se desarrolla en un mapa de 8x8 lugares. La orientación de los puntos cardinales es como se indica en la imagen anterior.

Diseño de la lógica del juego



En la lógica del juego tenemos la clase Game que se encarga de manejar la mecánica del juego. Decide cuándo empieza y cuándo termina una partida. También agrega jugadores a la partida.

Game tiene dos cronómetros (clase Timer). Uno de ellos se encarga de ejecutar la función que comienza la partida cuando la cantidad de jugadores es suficiente para ello. En caso de no poder, se reinicia. El otro cronómetro ejecuta la función que termina la partida y decide un ganador. Al terminar una partida, se genera un nuevo GameMap y el proceso anterior se reinicia para dar comienzo a un nuevo juego.

El ingreso de jugadores a la partida se hace de forma atómica, utilizando el objeto bloqueante gameAccess, para evitar que dos jugadores se coloquen en la misma posición o que hayan dos jugadores iguales en la misma partida.

Se usó eventos en varias clases para indicar sucesos importantes del juego.

La clase Game tiene un evento EndMatchEvent, el cual se ejecuta al terminar una partida. Con este evento se le puede avisar a todos los clientes que estén jugando que la partida ha terminado. También se le podría avisar a los jugadores que no están jugando que ya está disponible una nueva partida a la que pueden unirse (esto último no fue implementado).

La clase GameMap modela el mapa del juego. Esta maneja las posiciones de los jugadores con una matriz, cuya implementación oculta, exponiendo métodos de lectura y modificación. Los métodos de modificación son ejecutados bajo mutua exclusión, usando el objeto bloqueante mapEditionLock.
GameMap tiene un evento PlayerRemoved, el cual es utilizado para indicarle a los jugadores que han perdido la partida.

La clase Player modela los jugadores, los cuales pueden ser Monster o Survivor. Cada uno de ellos se encarga de modificar el mapa según el Role que tengan. En ningún momento es necesario realizar RTTI del Role del jugador gracias a que las clases Monster y Survivor implementan los métodos de modificación de mapa.
Para los diferentes tipos de ataques que posee el juego se utilizó el patrón Strategy, evitando el RTTI de roles.

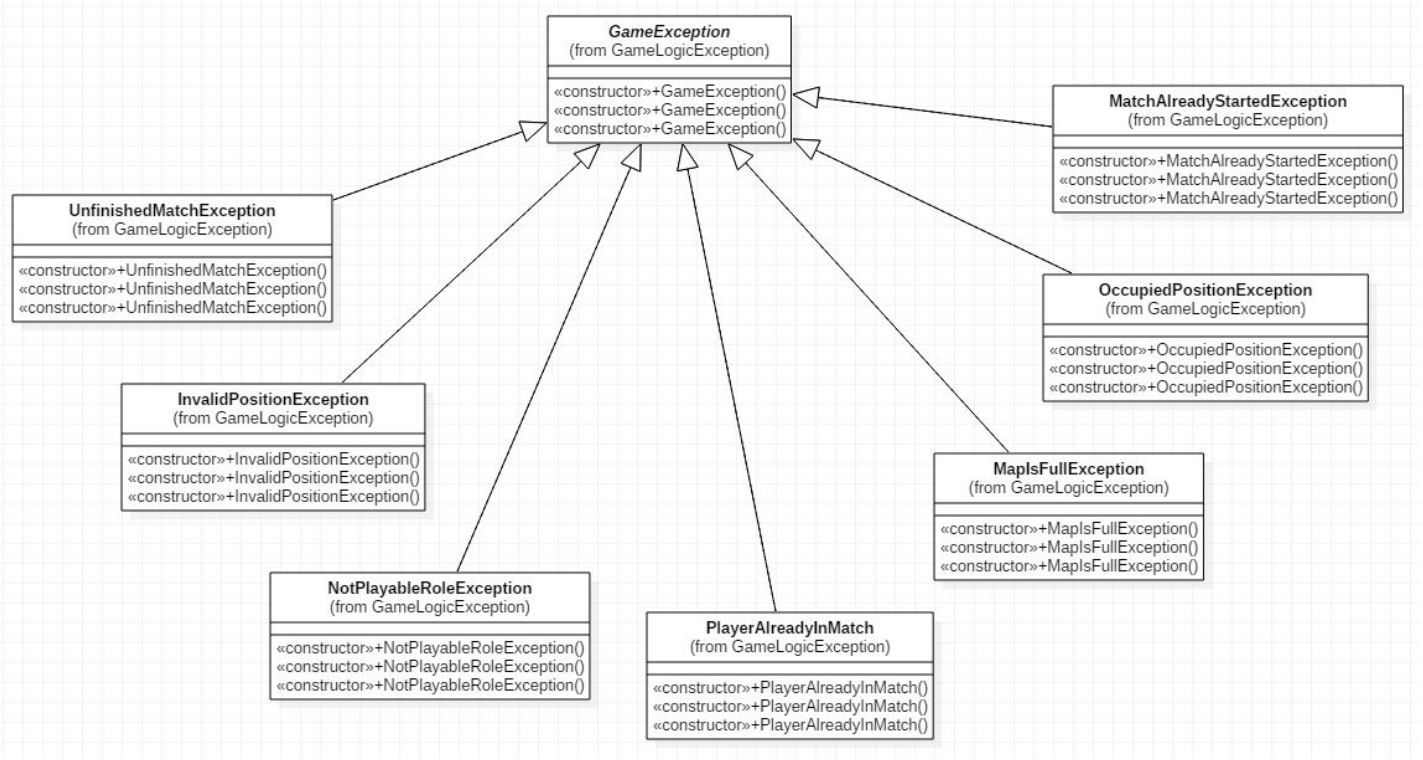
Player también posee un método Join para entrar a un Game. Cada clase hija se encarga de entrar como corresponda al juego.

Player posee los Action Notify y NotifyServer. Notify se utiliza para mandar notificaciones de un jugador y NotifyServer se utiliza para mandar notificaciones a un tercero que observa el juego (en este caso el servidor). Lo mejor para esto último hubiera sido tener solo un método de notificaciones pero se tuvo que agregar NotifyServer para cubrir un cambio de último momento rápidamente.

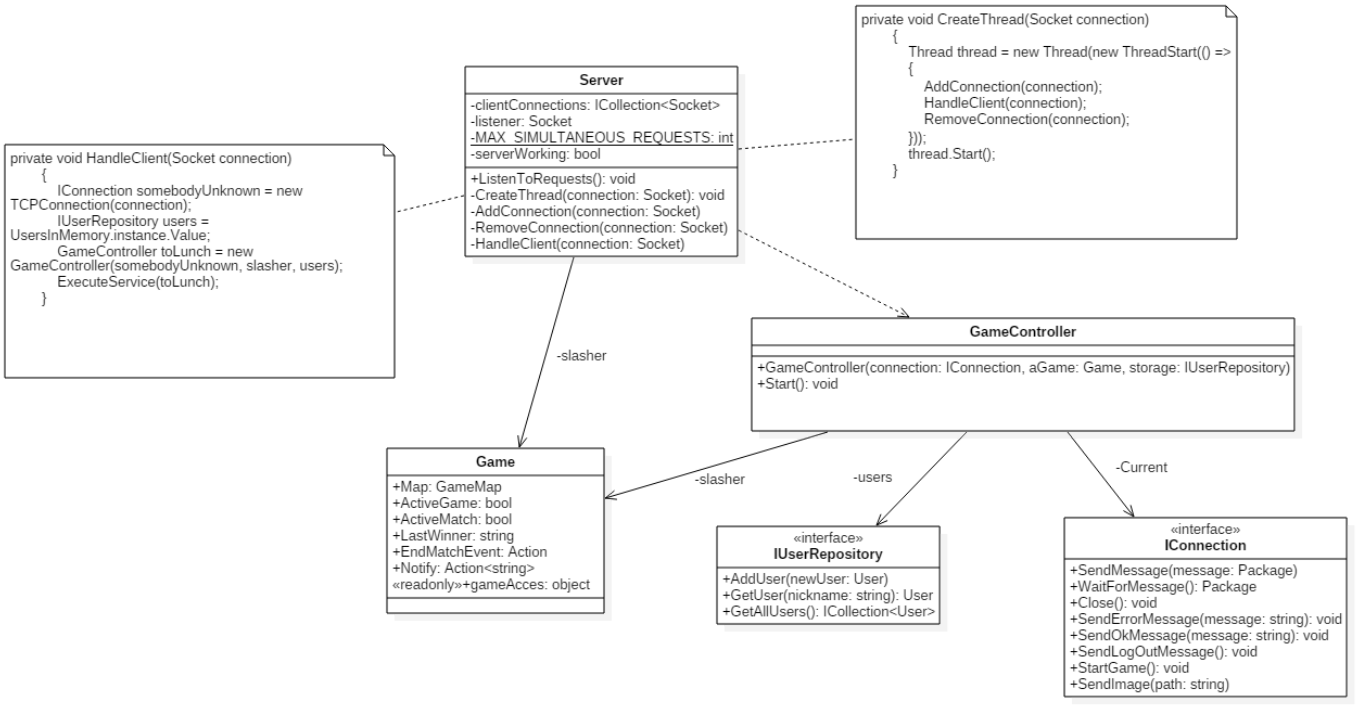
El movimiento de los jugadores es manejado por IMovementController, el cual maneja puntos cardinales y direcciones para determinar el movimiento de una Position a otra.

Los movimientos son forward o backward, los cuales se usan para determinar el movimiento de un jugador junto con la dirección en la que esté mirando.

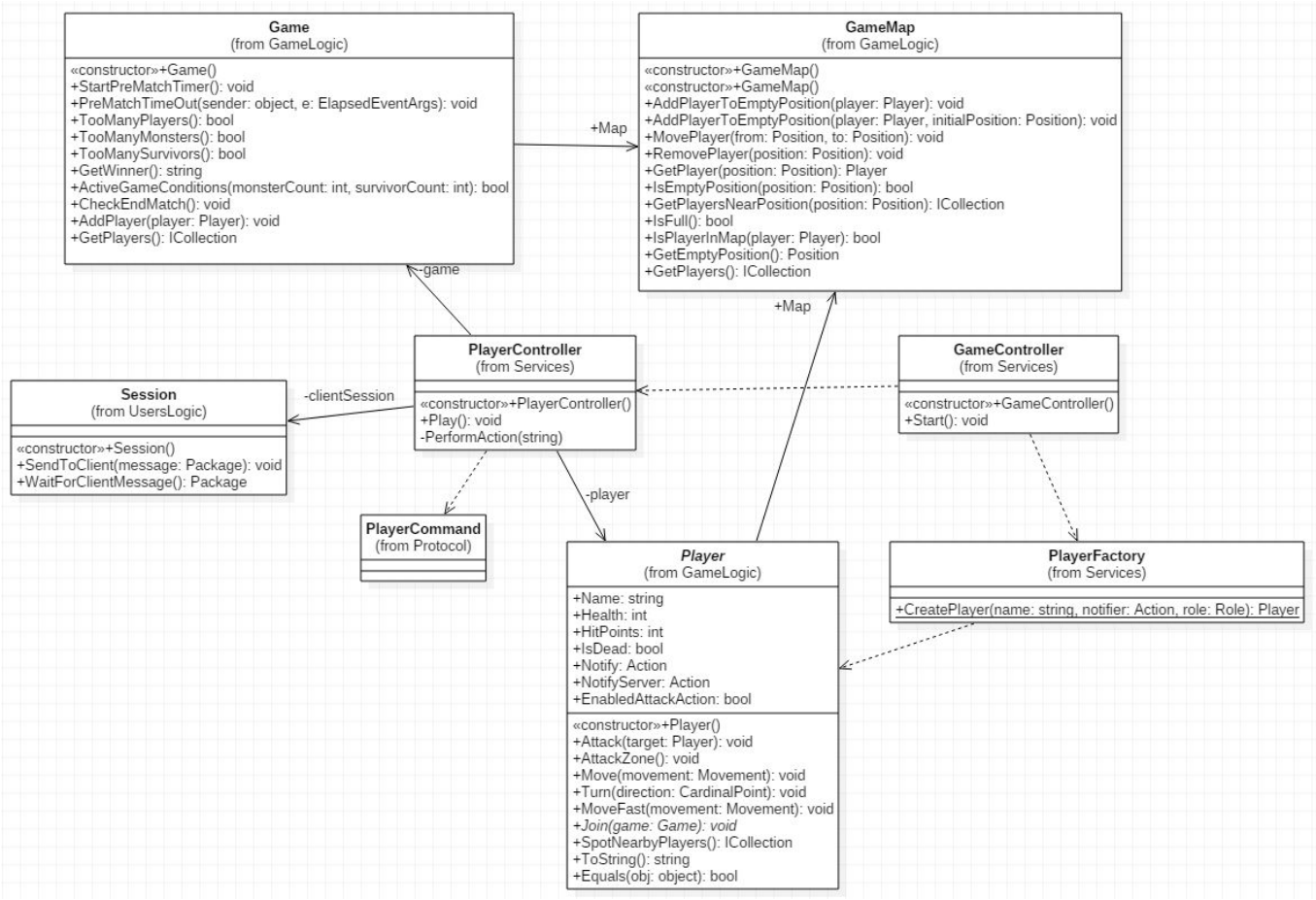
Las excepciones que se tiran desde la lógica del juego son las siguientes:



Diseño de controladores



Cuando un cliente se conecta al servidor, el servidor recibe la solicitud de conexión y lo redirige a un nuevo thread. En este nuevo thread, la clase GameController se encarga de recibir los request del usuario.
La conexión con el cliente se realiza en 3 etapas: crear la conexión, atender los request del cliente y remover la conexión.
La clase GameController se encarga de manejar los request del cliente. A la misma se le inyectan el Game del servidor, la conexión con el cliente y el repositorio de usuarios del servidor.
Cuando el cliente se loguea y entra al mapa, la atención de los request del cliente pasan a estar en manos de la clase PlayerController, la cual se encarga exclusivamente de atenderlo durante todo el desarrollo de la partida.



La creación de Player es hecha utilizando el patrón Factory debido a la complejidad de la clase. GameController es quien posee la información para crear el Player, por eso esa clase es quien se encarga de usar PlayerFactory.

GameController suscribe un método al evento Notify del player creado para informar al cliente de lo que le sucede a su jugador.
Una vez que el juego termina para el jugador, PlayerController termina su ejecución y se vuelve a GameController.

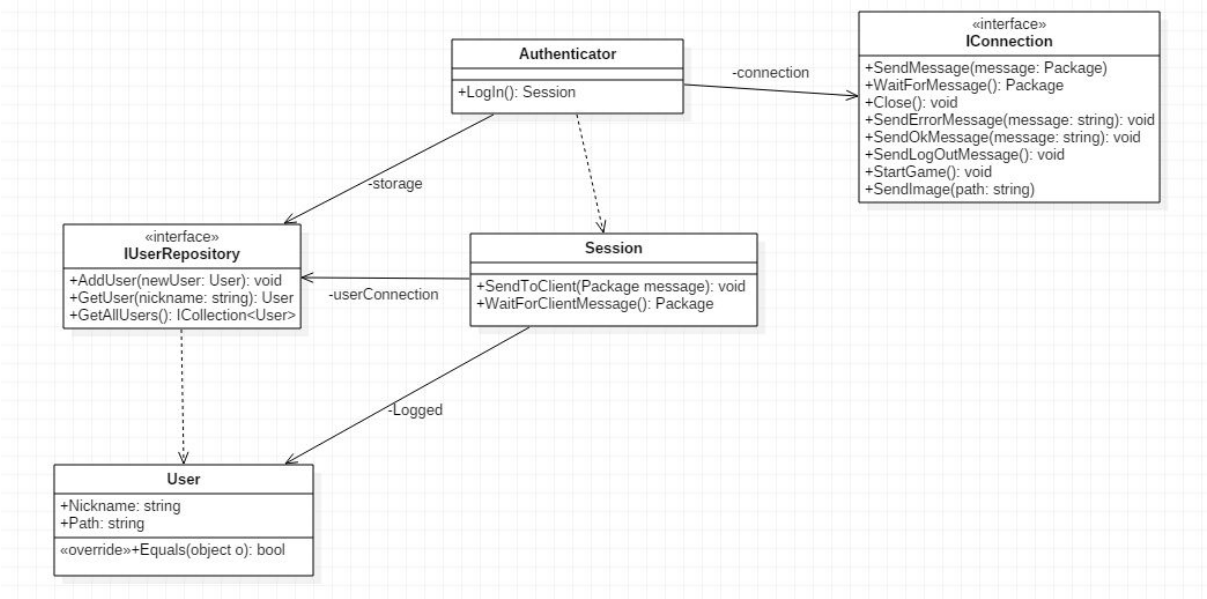
Los comandos de acciones están en una clase de constantes, PlayerCommand. De esta forma no se manejan literales en el código y se permite el cambio de los mismos sin impacto.

El motivo por el cual se usó eventos para los sucesos del juego fue la necesidad de mostrarle al cliente lo que pasa en el juego. Es por esto que desde el cliente se lanza otro thread que siempre escucha al servidor. De esta forma, las notificaciones son recibidas en tiempo real mientras el juego se sigue desarrollando.

Un problema que surgió fue que el único recurso de entrada y salida de datos que manejamos es la consola, por lo que ambos threads corriendo en el cliente debieron

acceder a ella bajo mutua exclusión. Para solucionar esto se creó la clase ConsoleAccess que regula el acceso al recurso compartido por ambos threads usando un objeto bloqueante.

Diseño de autentificación



El logueo de usuarios se realiza a través de la clase Authenticator. El método Login() espera por un mensaje de autentificación. Una vez obtenido, se obtiene el usuario del repositorio y en caso de existir y estar disponible, se le devuelve un objeto Session con este usuario. Esta operación se realiza bajo mutua exclusión para que no suceda que dos clientes se loguearon al mismo tiempo con el mismo usuario.

Authenticator es inyectado con IUserRepository y IConnection. Gracias a estas interfaces, se respeta el principio de inversión de dependencias, obteniendo así que la lógica del sistema no dependa de implementaciones de bajo nivel (como la conexión).

En este obligatorio, se modelaron los paquetes de la conexión TCP. El envío de paquetes se hace utilizando whiles que aseguren el correcto y completo envío de los mismos. La abstracción de la interfaz permite cambiar la implementación de la conexión a futuro en caso de que esto sea necesario.

Para el envío de la imágen, se optó por partir el archivo en varios paquetes en lugar de agrandar el campo del largo del paquete que viaja en el cabezal. Esto permite enviar imágenes de cualquier tamaño, sin agregar overhead (por el mayor tamaño del campo de largo del paquete).

Diagrama de Componentes