# 1 Diseño y evolución de las clases principales del Modelo

## 1.1 Diseño del tablero

Toda la lógica que implementa el concepto del tablero queda reflejada a lo largo de todo el proyecto en la clase Board.

### 1.1.1 Sprint 1

La clase Board es un simple contenedor de cubos, organizados en forma de matriz. No tiene interacción con otros objetos del modelo.

### 1.1.2 Sprint 2

Ahora la clase Board tiene la funcionalidad de actualizarse a sí mismo una vez se coloca un nuevo cubo, funcionalidad que antes estaba delegada a la clase Game.

### 1.1.3 Sprint 3

A partir de este Sprint los tableros tienen forma (es decir, la forma no es necesariamente siempre cuadrada) y tamaño elegido por los usuarios. A parte, Board tiene una representación en forma de String y de JSONObject, a través de los métodos toString() y report().

### 1.1.4 Sprint 5

Ahora, aparte de guardarse los cubos del tablero en forma de matriz, se guardan en forma de lista por ser una representación de los datos muy conveniente en distintas partes del proyecto, entre otras, para la red.

## 1.2 Diseño de los colores

## 1.3 Diseño de los cubos

## 1.4 Diseño del juego

La clase fundamental del juego es la clase Game, la clase principal del modelo.

### 1.4.1 Sprint 1

La clase Game responde a ejecuciones del PlaceCubeCommand, colocando nuevos cubos en las casillas seleccionadas, actualizando el tablero, manejando los turnos de los jugadores y actualizando sus puntos.

#### 1.4.2 Sprint 2

A partir de este momento, Game no se encarga de actualizar el tablero y las puntuaciones de los jugadores, sino de únicamente pasarle al tablero los cubos nuevos que tiene que insertar. El tablero, a partir de ese cubo, se actualiza a sí mismo, y los cambios de cubos actualizan las puntuaciones de los jugadores.

#### 1.4.3 Sprint 3

Game tiene su propia representación en forma de String y de JSONObject, a través de los métodos toString() y report().

#### 1.4.4 Sprint 4

A partir de este sprint, Game pasa a ser una clase abstracta y se pasa a tener dos modos de juego, el clásico (jugadores individuales) y el modo por equipos, que son implementados por las clases herederas de Game: GameClassic y GameTeams. También se implementa el patrón MVC, por lo que Game (modelo) pasa a tener una lista de observadores y métodos para el envío de notificaciones a estos.

#### 1.4.5 Sprint 5

Game se adapta con ciertos cambios y nuevos métodos para soportar el juego en línea.

#### 1.4.6 Sprint 6

En este momento Game sufre su mayor refactorización. Esta clase pasa a extender de la clase Thread, de forma que ya no funciona ejecutando comandos en el mismo momento de su creación, sino que estos de ponen en espera y Game, que está en todo momento funcionando y comprobando si hay nuevas peticiones puestas en espera, las ejecuta cuando puede. Esto nos permite evitar problemas de desbordamiento con el cálculo de jugadas por parte de las inteligencias artificiales, aparte de permitir el funcionamiento esperado de la vista sin comprometer su rendimiento. Aparte, Game deja de llevar a cabo el manejo de turnos y se delega esa responsabilidad a la clase TurnManager, que es invocada tras cada jugada para que ejecute (si procede) el siguiente turno.

## 1.5 Diseño de los jugadores

## 1.6 Diseño de los equipos

## 1.7 Diseño del gestor de turnos

## 1.8 Diseño de los estados del juego

## 1.9 Diseño de las replays

## 1.10 Diseño de las Inteligencias Artificiales

### 1.10.1 Sprint 5

Las estrategias son externas al modelo. El cómputo de movimientos a través de estas estrategias de hace a partir de la vista, a través de la clase PlayerView, que representa al Player en la vista y se encarga de ejecutar sus acciones. Por razones de encapsulación, las estrategias no tienen acceso al modelo y realizan sus cálculos a través de GameStates.

### 1.10.2 Sprint 6

Las estrategias ahora forman parte del modelo, siendo atributo de aquellos Players controlados por la máquina. Como en este punto el manejo de turnos se lleva a cabo por la clase TurnManager y el modelo funciona en su propia hebra, la clase PlayerView desaparece del proyecto y son los propios players quienes ejecutan las estrategias, que a nivel abstracto resulta mucho más intuitivo. Por la hebra del modelo, cuando las estrategias terminan de calcular el siguiente movimiento, este no se ejecuta en ese mismo instante, sino que se deja en espera en el modelo hasta que este pueda ejecutarlo.

# 2 Diseño del Controlador

# 3 Diseño de la Vista de GUI

## 3.1 Diseño del menú principal y pantallas pre-juego

## 3.2 Diseño de la pantalla de juego

# 4 Diseño de la Vista de Consola

## 4.1 Diseño del menú principal y pantallas pre-juego

## 4.2 Diseño de la pantalla de juego

# 5 Diseño de la red

## 5.1 Diseño del servidor

## 5.2 Diseño de los clientes

# 6 Diseño y evolución de las Historias de Usuario

## 6.1 Como usuario quiero que Rolit introduzca características innovadoras pensando en las posibilidades que brinda el multijugador: Inteligencias Artificiales

### 6.1.1 Sprint 5

Este fue el Sprint en el que se empezaron a desarrollar las distintas estrategias de las inteligencias artificiales. Se planearon tres, recogidas en las siguiente clases, todas herederas de la clase abstracta Strategy: RandomStrategy, GreedyStrategy y MinimaxStrategy.

La idea de la estrategia es que, cuando le toque jugar a una inteligencia artificial, la estrategia se encarge de calcular su siguiente movimiento y este se ejecutase inmediatamente después de su cálculo.

Para encapsular está lógica se creo la clase abstracta Strategy, para que cada estrategia en particular fuera una clase heredera de esta.

Se han desarrollado tres estrategias, que suponen tres niveles de dificultad distintos, y la lógica de estas está recogida en las siguientes clases: RandomStrategy, GreedyStrategy y MinimaxStrategy.

**RandomStrategy**: La idea es que se genere una posición cualquiera en el tablero, siempre y cuando esta sea válida. Esta es la posición que la inteligencia artificial jugará. Lógicamente, la tendencia general las inteligencias artificiales que aplican esta estrategia es no obtener una gran cantidad de puntos, por lo que esta estrategia es la de nivel fácil.

**GreedyStrategy**: Esta estrategia tiene por intención analizar el tablero en busca de la posición que le garantiza al jugador el máximo número de puntos en

este mismo turno. Esta estrategia lleva a jugadas mucho mejores y elaboradas, pero sigue sin ser la mejor, así que representa el nivel de dificultad medio.

**MinimaxStrategy**:

Antes de explicar la implementación de esta estrategia en el caso de Rolit, debemos explicar primero en qué consiste la estrategia Minimax en teoría de juegos:

**Estrategia Minimax en teoría de juegos**:

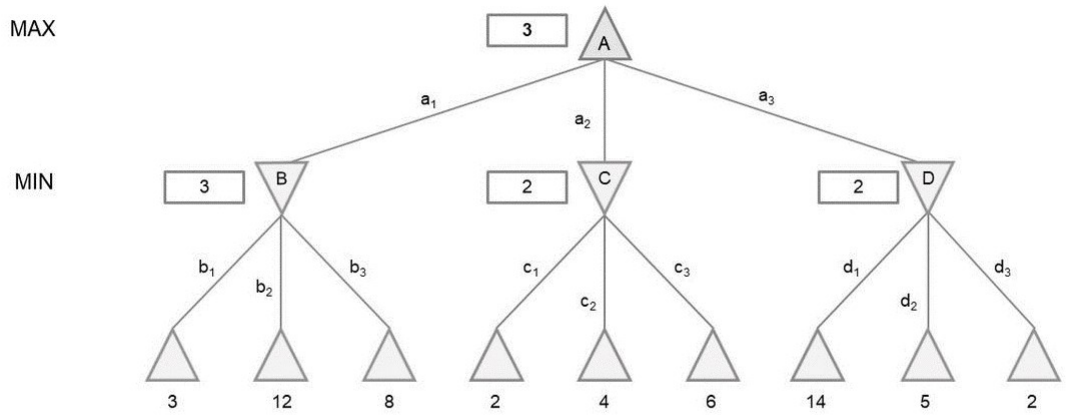
En teoría de juegos, el Minimax busca minimizar la pérdida esperada. La aproximación que se toma es asumir que el oponente va a tomar las decisiones que más te perjudiquen. De esta manera, al encontrar la decisión que menor pérdida suponga, el resultado real será siempre igual o mejor al calculado, de forma que el cálculo es fiable.

La mejor forma de explicar esto es a través de un ejemplo:

Imaginemos que estamos en un juego de dos jugadores, uno contra el otro, basado en turnos, en el cual ambos jugadores conocen en todo momento el estado actual de la partida en su totalidad. Un buen ejemplo de esto es el ajedrez. Supongamos pues que juegas con las piezas blancas, y tu adversario juega con las piezas negras. En cada uno de tus movimientos vas a jugar el movimiento que consideres que más te favorece. Por el otro lado, bajo nuestra aproximación, suponemos que el otro jugador va a jugar el movimiento que más te perjudique. Podemos hacer una representación de esto en forma de árbol:

Imaginemos que cada nodo contiene un número que representa el estado actual de la partida (para esto hace falta tener un criterio de valoración del estado actual de la partida, en el cual no entraremos en detalle en el caso del ajedrez, pero más adelante sí en el caso del Rolit), y cada arista representa un movimiento jugado, por el cuál se desciende en el árbol de un estado de la partida al siguiente. En cuanto a la valoración de los estados del juego, si el número de un nodo es positivo va ganando el jugador blanco (a mayor mejor); si el número es negativo, va ganando el jugador negro (a menor peor). Como el juego va por turnos, si un nivel del árbol se corresponde con el turno de un jugador, el siguiente nivel se corresponde con el siguiente jugador. De esta forma, volviendo al ejemplo propuesto, si desde un nodo se conoce el valor de todos sus descendientes pueden pasar dos cosas: si es el turno del jugador blanco, tomará la decisión que le lleve al mayor valor; si es el turno del jugador negro, tomará la decisión que le lleve al menor valor.

Se ilustra el funcionamiento del algoritmo en la siguiente imagen:




En el caso del ajedrez (y de la mayoría de juegos por turnos, como Rolit) hay siempre muchos posibles movimientos a jugar. De esta forma, en el árbol de decisión, de cada nodo salen muchos descendientes, resultando en un algoritmo con coste aproximadamente exponencial en el promedio de jugadas disponibles. Esto hace que la búsqueda del mejor posible movimiento se convierta en un problema intratable en no demasiados niveles de profundidad de búsqueda. Por tanto, surge la obligación de limitar la profundidad hasta la que se quiere hacer la búsqueda.

Volviendo ahora al Rolit, no estamos en un juego de dos jugadores (o al menos no necesariamente). Afortunadamente, en este juego el criterio de valoración de jugadas es fácil: La mejor jugada es la que te lleve a acabar con el mayor número de puntos.

Ahora, como hay más de dos jugadores, para conseguir un cálculo de puntos fiable, el cálculo se lleva a cabo considerando que el jugador propietario de la estrategia quiere hacer aquella jugada que más puntos le otorgue a la larga, mientras que el resto de jugadores en sus turnos hacen la jugada que más puntos le quite al jugador propietario. De esta forma se calculan puntos para el jugador propietario situándonos en la situación más desfavorable posible, de forma que todo aquello que se calcule va a derivar siempre en un resultado igual o mejor al calculado, de forma que los resultados de los cálculos son fiables.

Debido a lo exhaustivos que resultan estos cálculos (y tras comprobación empírica simulando numerosas partidas) la estrategia MinimaxStrategy representa el nivel difícil de las inteligencias artificiales.

Ahora, antes de la explicación más técnica, observemos que la estrategia

GreedyStrategy se puede implementar aplicando una MinimaxStrategy en la que solo se explora un nivel de profundidad, puesto que en este nivel se busca la jugada que más puntos garantice, y no se sigue buscando más allá. Por tanto, a nivel de clases, la clase GreedyStrategy es heredera de MinimaxStrategy, y el atributo de profundidad máxima pasa a valer 0.

Pasamos ahora a explicar la implementación de estas estrategias:

Para la simulación de movimientos pensamos originalmente en usar la clase Board para colocar cubos y evaluar resultados, pero no tardamos en darnos cuenta de que esto era inviable, puesto que los métodos de Board tienen una comunicación con otras clases que no deseeamos para esto, puesto que nosotros simplemente queremos hacer simulaciones, y no cambios reales.

Es por esto que fue necesario crear otra representación del tablero puramente funcional y adaptada a la simulación de movimientos, de donde surgió la siguiente clase:

**SimplifiedBoard**:

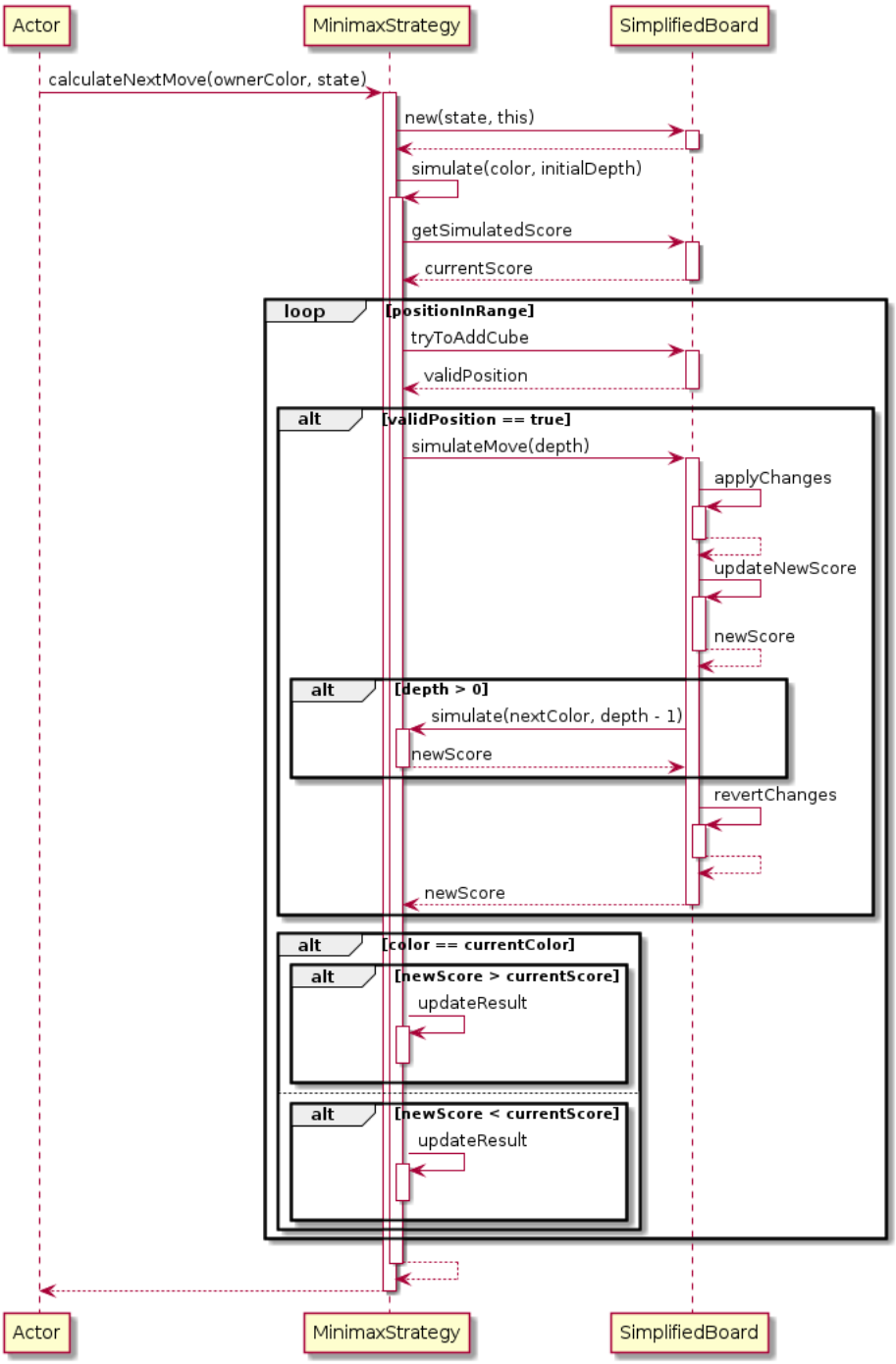
Esta clase consta de una matriz en la que almacena el color de los cubos del tablero real. Para disminuir costes y evitar tener que hacer copias del tablero tras cada movimiento simulado, se lleva una pila con los cambios que se realizan al simular un movimiento, de forma que cuando se quiere dejar el tablero en el estado previo a la simulación para realizar otra simulación, en vez de realizar una copia se revierten los cambios aplicados, lo cual resulta mucho menos costoso.

En SimplifiedBoard también se almacenan los puntos de los distintos jugadores, puesto que la idea es consultar los puntos después de simular cada movimiento.

Ahora, para calcular el mejor movimiento para ejecutar, en la clase de la estrategia se realiza un bucle en el que se recorren todas las posiciones del tablero, consultando si cada posición es válida o no, y en caso de dar con una posición válida, se simula ese movimiento.

Dentro de la simulación, en SimplfiedBoard, si la profunidad a explorar es mayor que 0, antes de revertir los cambios se vuelve a realizar el bucle de las posiciones, pero simulando esta vez para el siguiente jugador, y así hasta que la profundidad a explorar es 0. Hay que tener en cuenta que el jugador propietario de la estrategia busca maximizar sus puntos, mientras que el resto de jugadores buscan minimizarlos. Por tanto, en los bucles de recorrido de posiciones, la estrategia es conocedora de para qué jugador esta simulando el siguiente movimiento, de forma que si está simulando para el jugador propietario devolverá el resultado más favorable, y si está simulando para cualquier otro jugador devolverá el resultado más perjudicial posible para el propietario. De esta forma, se podrá conocer el resultado final realista de cada jugada posible, y así elegir la mejor jugada para el jugador propietario.

El cómputo del movimiento a jugar a través de la estrategia Minimax, llevado a cabo en el método calculateNextMove(), se ilustra en el siguiente diagrama:


Actor
MinimaxStrategy
SimplifiedBoard
calculateNextMove(ownerColor, state)
new(state, this)
simulate(color, initialDepth)
getSimulatedScore
currentScore
loop
[positionInRange]
tryToAddCube
validPosition
alt
[validPosition == true]
simulateMove(depth)
applyChanges
updateNewScore
newScore
alt
[depth > 0]
simulate(nextColor, depth - 1)
newScore
revertChanges
newScore
alt
[color == currentColor]
alt
[newScore > currentScore]
updateResult
alt
[newScore < currentScore]
updateResult
Actor
MinimaxStrategy
SimplifiedBoard

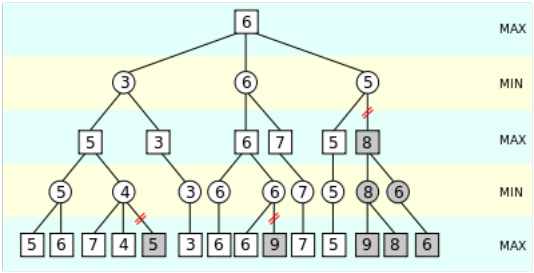
### 6.1.2 Sprint 6

Por motivos de eficiencia de MinimaxStrategy, se ha implementado de forma complementaria la poda alfa-beta, que se explica a continuación:

**Poda alfa-beta**: La poda alfa-beta es una mejora del algoritmo Minimax. Se mantienen dos valores, alfa y beta, que representan respectivamente la puntuación mínima que se llevará el jugador maximizador y la puntuación máxima que se asegura el jugador minimizados. Inicialmente, alfa es $-\infty$ y beta es $\infty$.

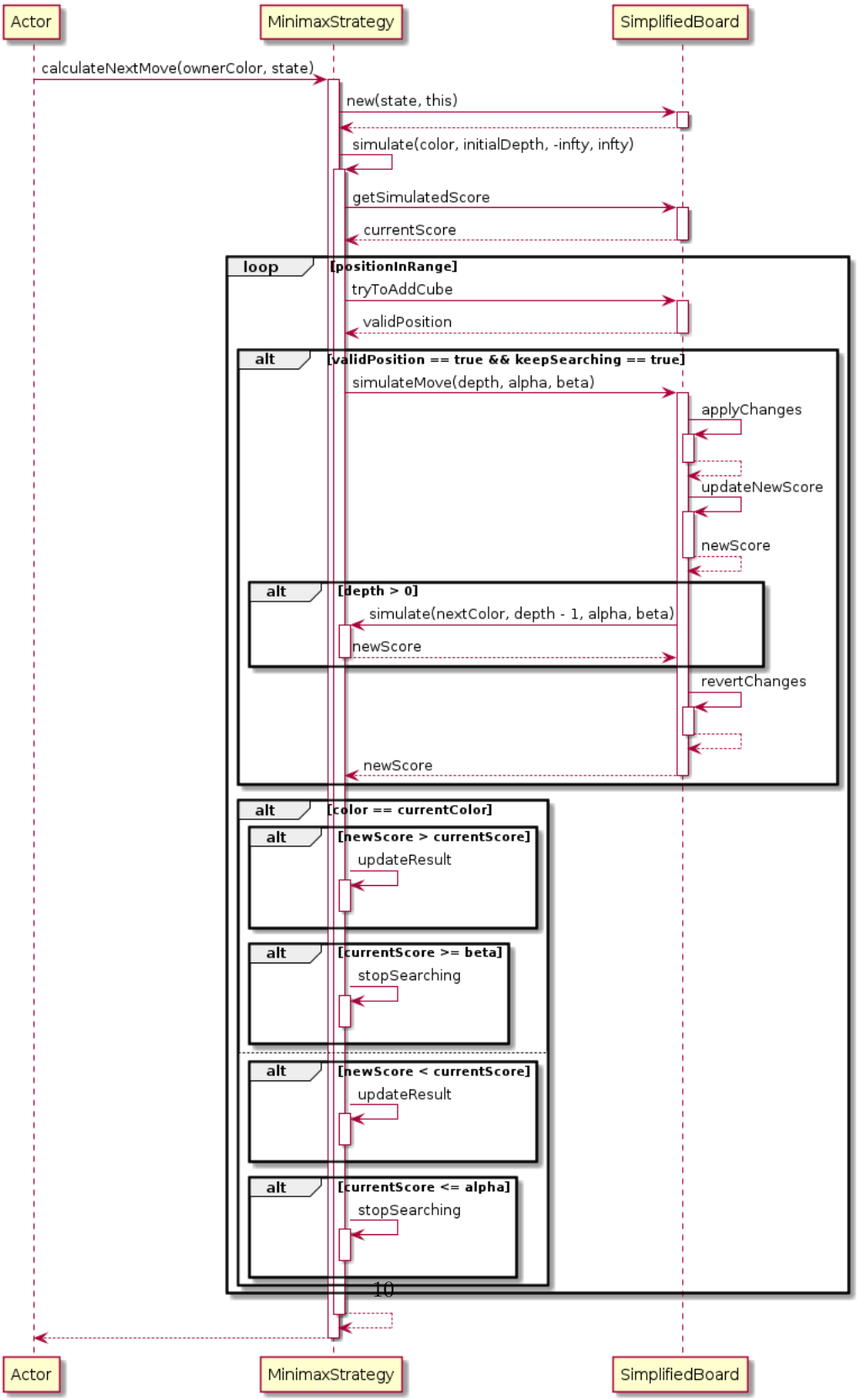
Siempre que la puntuación máxima que se asegura al jugador que minimiza se vuelve menor que la puntuación mínima que se asegura el jugador que maximiza se puede parar de explorar por la rama actual. De forma análoga, se podan ramas en el caso contrario.

La mejor forma de visualizar esta poda es a través de una ilustración:




Como vemos en este ejemplo, si exploramos en el árbol de izquierda a derecha, una verz llegamos a la rama derecha vemos que el jugador minimizador encuentra una rama por la que logra llegar al valor 5. Como minimiza, se sabe que el valor de ese nodo va a ser, como mucho, 5. Al ver esto el jugador maximizador, teniendo en cuenta que en una rama anterior ha llegado al valor 6, sabe que no tiene que seguir explorando esa rama, porque de ninguna manera va a encontrar un valor mejor que 6, y por tanto, en esta situación, el mejor resultado es el que le brinda empezar explorando la rama del centro.

Esta poda ha sido muy útil para reducir costes de cálculo, y el nuevo algoritmo mejorado queda reflejado en el siguiente diagrama:


Actor
MinimaxStrategy
SimplifiedBoard
calculateNextMove(ownerColor, state)
new(state, this)
simulate(color, initialDepth, -infty, infty)
getSimulatedScore
currentScore
loop
[positionInRange]
tryToAddCube
validPosition
alt
[validPosition == true && keepSearching == true]
simulateMove(depth, alpha, beta)
applyChanges
updateNewScore
newScore
alt
[depth > 0]
simulate(nextColor, depth - 1, alpha, beta)
newScore
revertChanges
newScore
alt
[color == currentColor]
alt
[newScore > currentScore]
updateResult
alt
[currentScore >= beta]
stopSearching
alt
[newScore < currentScore]
updateResult
alt
[currentScore <= alpha]
stopSearching
Actor
MinimaxStrategy
SimplifiedBoard

## 6.2 Como usuario, me gustaría jugar a Rolit pensando en las posibilidades que brinda el multijugador (Red)

### 6.2.1 Sprint 5

En cuanto a la tarea de añadir un modo de juego en red, se concibe, planea e implementa la funcionalidad de red casi por completo, consigue llegar a una versión funcional de juego en red en GameClassic. Al final del Sprint, los objetivos alcanzados son los siguientes:

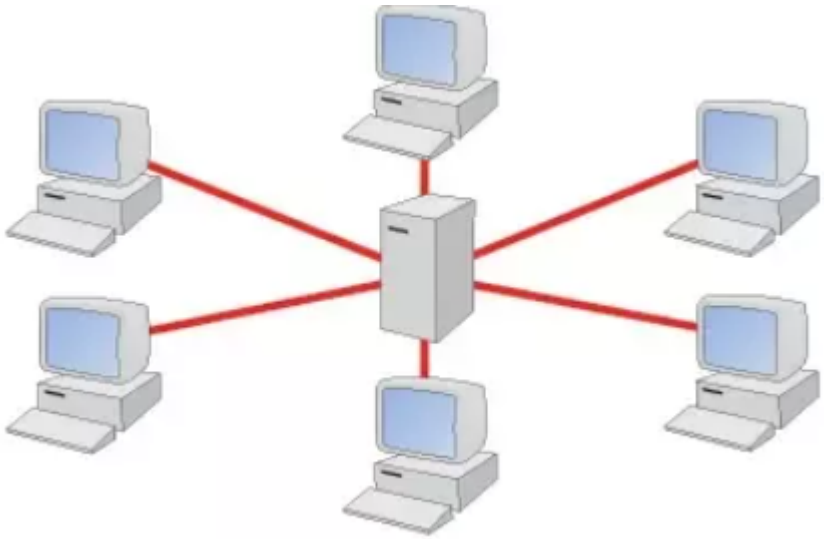
- Determinar si el servidor, o por el contrario el cliente, debería poseer el modelo. Optamos por la segunda opción.
- Definir toda la estructura en cuanto a relaciones jerárquicas de clases y dependencias entre las mismas.
- Crear un número suficiente de diálogos en GUI que permitan la conexión en red. Entre ellos, se encuentran:
  - ServerView
  - JoinServerDialog

  El diseño y evolución pormenorizados de estos diálogos se encuentran en el apartado de la historia de usuario dedicada a la interfaz.
- Reutilizamiento del código del diálogo de crear partida, adaptado a las circunstancias de red.
- Poder jugar a una partida GameClassic en red de forma satisfactoria.

Aun así, otros de los objetivos propuestos no son implementados por falta de tiempo y de dependencia con otras de las partes del desarrollo no concluidos. Estos objetivos son:

- Crear más diálogos que aporten feedback para la conexión, tanto de la perspectiva del cliente como del servidor.
- Llegar a una versión completamente refactorizada y con métodos simplificados.
- Implementar el juego en red en GameTeams.

Fundamentalmente, la conexión en red se basa en una estructura en el que los clientes poseen el modelo, realizan cambios en el mismo y lo notifican al servidor. El servidor procede a enviar al resto de clientes la información nueva según la cual deben actualizar sus modelos. Nótese que el cliente sólo tiene contacto con el servidor, y el servidor tiene contacto con todos los clientes. Es, por tanto, un modelo de red centralizado, con nodo central el servidor.



Para implementar esta funcionalidad de red en Java operamos según el modelo ServerSocket-Socket. Primero, desde la perspectiva del usuario que abre el servidor, se debe crear una instancia de ServerSocket pasándole como parámetro en el constructor el puerto en el que localmente debe operar el servidor. Posteriormente, se crea un nuevo Socket por medio de llamar al método accept() de ServerSocket. De esta forma, tenemos un socket asociado a un cliente en específico. Para n clientes el servidor necesitará n sockets. Cada cliente solo necesita un socket, pues solo tiene conexión con el servidor y no con el resto de clientes.

```
//Perspectiva del servidor

ServerSocket serverSocket = new ServerSocket(port);
Socket socketCliente1 = serverSocket.accept();
...
...
...
Socket socketClienteN = serverSocket.accept();
```

Este método accept() se queda "bloqueado" o en espera, hasta que un cliente se conecta por medio de la creación de un Socket desde su aplicación de la siguiente manera:

```
//Perspectiva del cliente

Socket socket = new Socket(ip, port);
```

donde ip es la dirección IP donde opera el servidor (ya sea local, o pública con el puerto especificado abierto para permitir conexiones desde fuera de su red), y port el puerto donde esta opera.

Una vez que las creaciones del Socket de servidor y cliente han sido creadas de forma satisfactoria, la conexión ha sido realizada con éxito. Por tanto, podemos proceder al envío de mensajes entre cliente-servidor y viceversa.

Para ello, Java nos ofrece realizar este cambio de información por medio de pares de instancias BufferedReader-PrintWriter, llamémosle in e out respectivamente. Estas instancias leen y reciben String, respectivamente. Nuestro modelo procederá a enviar JSONObject pasados a formatos String, que después al ser recibidos como Strings se volverán a construir en JSONObject por medio de la constructora de JSONObject que admite como parámetro un String.

Cada servidor posee un número de parejas in-out equivalente al número de clientes conectados, y cada cliente posee una pareja.

```
//Perspectiva del servidor
BufferedReader inCliente1 = new BufferedReader(new
InputStreamReader(socketCliente1.getInputStream()));
...
...
...
PrintWriter outClienteN = new PrintWriter(
socketClienteN.getOutputStream(), true);
```

```
//Perspectiva del cliente
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(
socket.getOutputStream(), true);
```

De esta forma, cada vez que se pretende enviar un mensaje desde el cliente al servidor, el cliente k envía un mensaje msg por medio del método out.println(msg). Este mensaje es recogido desde el servidor por el método inClienteK.readline(). Asimismo, si el servidor pretende enviar un mensaje al cliente k, este debe llamar a outK.println(msg). El cliente k recoge el mensaje desde in.readline(). En resumen:

```
        //Envio de mensaje ClienteK Servidor


//Perspectiva del cliente k
out.println(msg);

---->

//Perspectiva del servidor
String msg = inClienteK.readline();
```

```
        //Envio de mensaje Servidor-ClienteK


//Perspectiva del servidor
outK.println(msg);

-->

//Perspectiva del cliente k
String msg = in.readline();
```

Sin embargo, surgen una serie de problemas con respecto a este modelo. El servidor debería recibir de manera independiente, paralela y en cualquier momento los peticiones de cada cliente, de lo contrario habría que definir un orden completamente arbitrario de llegada de mensajes según cliente. Si bien esto pudiera hacerse para realizar juegos por turnos en el que un mensaje enviado por un cliente al que no le corresponde el turno no sea procesado por el servidor, otras funcionalidades serían imposibles de implementar.

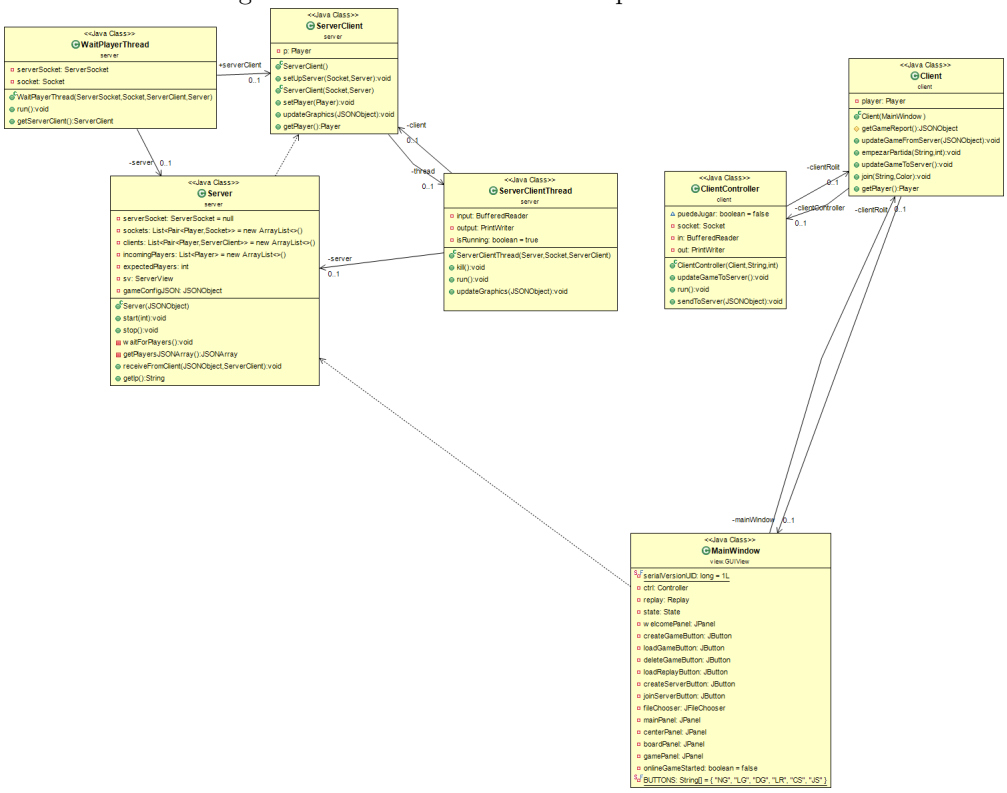
Por ejemplo, podría ocurrir que en el modo por equipos, dos clientes se conecten. Reciben ambos del servidor una lista de equipos en la que pueden conectarse. Se ve que es absurdo imponer un orden de quién envía la información del equipo elegido, pues esta es imposible de anticipar. Puede darse que el primer cliente que se conecte sea el segundo que especifique en qué equipo quiere conectarse.

En suma, nos interesa que la llegada de mensajes no deba ser regida por un orden definido y arbitrario. Para ello, nos interesa utilizar una herramienta de la programación concurrente y paralela, los hilos; en el caso de Java, proporcionados por la clase Thread.

De esta manera, si el servidor tiene un hilo por cada cliente, podrá recibir mensajes de forma paralela. Así, ningún mensaje enviado desde un cliente es perdido; todos llegan al servidor con independencia de cuándo se emitan desde el cliente. Desde la perspectiva del servidor, este es el cometido de la clase ServerClientThread implementada en el proyecto, que extiende de Thread. En su método run, recibe mensajes con el método readLine() anteriormente descrito **de forma periódica y constantemente (en un while), hasta que se haya decidido cerrar el juego**. Posteriormente, se envía este mensaje al método receiveFromClient, synchronized (pretendemos ejecutar múltiples procesamientos que de forma secuencial para evitar errores imprevistos), que pertenece a la clase Server, que de forma sincronizada procesa este mensaje.

El cliente se encuentra en una situación parecida. ¿Qué ocurre si recibe una información del servidor, la procesa y mientras se da este procesamiento, recibe otro mensaje del servidor? Necesitará el cliente, por tanto, dos hilos: un hilo que se dedique a recoger mensajes, y otro hilo que se dedique a procesarlos. De esta forma, ninguna información emitida desde el servidor es perdida.

Una vez el modelo de red ha sido completamente explicado, procedemos a detallar los detalles de implementación relativos al juego en específico. Es preciso introducir el diagrama de clases relativo a este Sprint.




Empezamos a detallar la estructura del servidor.

ServerView, especificado en el apartado de diseño correspondiente, tiene el papel de pasar la información pertinente para el funcionamiento de la clase Server.

Partimos de la clase Server, clase que gestiona los clientes desde la perspectiva del servidor y que procesa los mensajes emitidos desde los clientes. Como hemos anticipado, Server debe poseer un único ServerSocket a través del cual se abre el servidor. Posee el atributo expectedPlayers recibido desde la GUI, en el que el usuario que ha abierto el servidor especifica qué número de jugadores se conectarán al servidor. Es de vital importancia conocer este dato, pues de ello dependerá el número de WaitPlayerThread creados, clase que pasaremos a comentar después.

Asimismo, Server posee dos listas: una correspondiente a pares Player-Socket, de modo que a cada jugador se le asocia el Socket a través del cuál puede comunicarse con el cliente en específico que juega bajo su identidad; y otra correspondiente a pares Player-ServerClient, donde a cada jugador se le asocia el ServerClient específico. Pasaremos a describir posteriormente qué es la clase ServerClient. En resumen, tenemos asociaciones biunívocas jugador-

cliente las cuales aprovecharemos para el envío y la recepción de mensajes.

El ServerClient constituye una representación de un cliente en específico desde la perspectiva del servidor.

Lo fundamental de la clase ServerClient es que posee el thread encargado de la recepción directa de mensajes desde el cliente en específico; es decir, posee una única instancia de clase ServerClientThread, anteriormente mencionada. ServerClient posee una referencia a su ServerClientThread y viceversa. Esto es porque:

- ServerClient posee a ServerClientThread porque el servidor, al enviar mensajes al cliente, pasa el mensaje por ServerClient (recordemos la lista Player-ServerClient) quien pasa a enviárselo a a ServerClientThread.
- ServerClientThread posee a ServerClient porque al recibir mensajes desde el cliente, ServerClientThread envía el mensaje a Server para procesarlo. En esta función server.receiveFromClient, se necesitan dos parámetros: el mensaje, y el ServerClient asociado. Es por esto que para este segundo parámetro, ServerClientThread necesite una referencia de ServerClient.

Como vemos, la finalidad es encapsular el código de forma que Server no conozca de ServerClientThread sino solo de ServerClient.

WaitPlayerThread, por otra parte, es un hilo encargado de ir recibiendo los jugadores. Se crean un número "expectedPlayers" (atributo de Server) de instancias de esta clase, cada una de ellas con un ServerClient asociado (por tanto, ServerClient es atributo de WaitPlayerThread). En su método run se encarga de ejecutar el método serverSocket.accept() que especificamos anteriormente. En cuanto se acepta la conexión, se procede a llamar al método setUpServer de ServerClient para que este cree su ServerClientThread.

La necesidad de hacer esto de forma paralela justifica la creación del hilo WaitPlayerThread. Para evitar problemas de concurrencia, en esta clase los atributos Server y ServerClient son volátiles para que todos los hilos WaitPlayerThread conozcan en tiempo real el estado de Server y ServerClient.

En particular, Server debe ser volatile pues debe ir actualizando su lista de Player-Socket en tiempo real y de forma organizada de a fin de evitar bugs. Esto es porque el método sincronizado waitForPlayers de Server recoge periódicamente el número de conexiones aceptadas. En cuanto estas conexiones igualan el número de conexiones aceptadas, el método deja de ejecutarse y se procede a las gestiones que tiene que realizar el servidor una vez todos los usuarios esperados se han conectado.

Una vez detallados los cometidos de todas las clases que posee el servidor, pasamos a detallar las del cliente.

El punto de partida es la clase Client, que será un intermediario entre su thread de recepción de mensajes y la GUI. Por ello, MainWindow precisa una referencia de Client y Client una de MainWindow; es decir, la comunicación es bidireccional. La razón de esto es:

- Al hacerse una jugada en el modelo, como se ha especificado, necesitamos pasar el nuevo estado del juego al servidor para que este, a su vez, se la

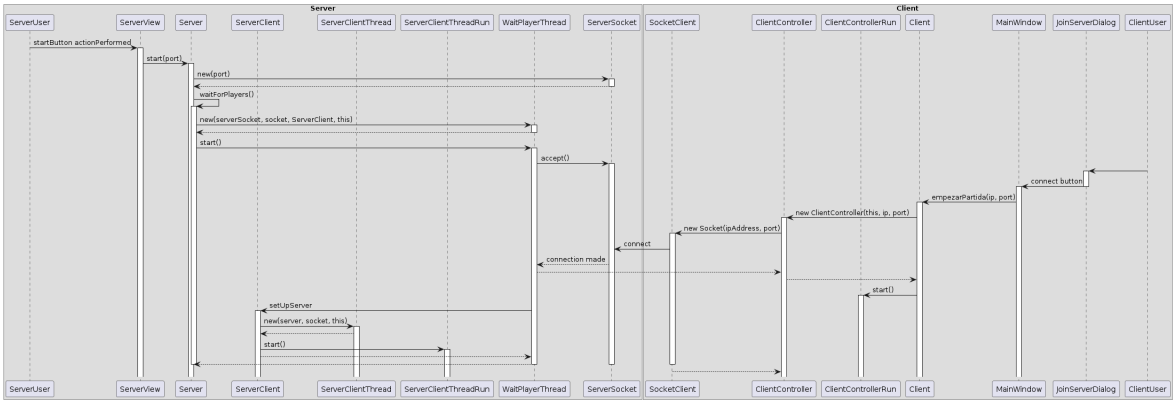
pase al resto de clientes. Por tanto, MainWindow pasa el estado del juego a su Client. MainWindow, a su vez, obtiene este estado del juego desde el controlador. Todos estos pasos de información se realizan a través de los métodos updateGameToServer que poseen estas clases.

- Al recibirse un nuevo estado del juego desde el servidor, Client recibe esta información desde su thread. Necesita a MainWindow para que este, a su vez, envíe el nuevo estado del juego al controlador con el fin de actualizar el modelo al nuevo juego requerido. Todos estos pasos de información se realizan a través de los métodos updateGameFromServer que poseen estas clases.

Finalmente, pasamos a describir ClientController, el thread de Client. Nuevamente la comunicación entre ambas clases es bidireccional: para enviar información al servidor, Client notifica a ClientController; para recibir información del servidor, ClientController notifica a Client.

Como hemos anticipado, ClientController extiende de Thread, y en su método run() recibe información del servidor que procede a enviar al cliente. El método sendToServer envía la información directamente al servidor por medio de out.println como hemos descrito.

Una vez especificadas todas las clases, describiremos el hilo típico de la ejecución para comprender cómo operan estas y en el orden en el que lo hacen.




Observamos que al pulsar el botón de Start Server en el diálogo de abrir un servidor, se llama crea Server pasándole como parámetro un JSON (gameConfigJSON) correspondiente a una configuración del juego básica especificada desde la GUI, que el servidor rellenará añadiendo los jugadores pertinentes. En el constructor de Server se almacena este atributo JSON y se llama al método start(port) que crea el ServerSocket. Posteriormente, llamada al método waitForPlayers para realizar las conexiones oportunas. Este método se compone de un bucle for que recorre un número de vueltas equivalente al del número de jugadores a conectarse esperados. por cada vuelta, se crea su ServerClient y Client asociados, y se crea el thread WaitPlayerThread y se llama a start(). El WaitPlayerThread, en su método run() llamado desde start(), espera en el método de serverSocket.accept() que se conecte un cliente.

Un cliente, por medio de JoinServerDialog, se conecta al servidor desde la IP y puerto requeridos. Pasa por el MainWindow quien a su vez llega al cliente (Client) con el método empezarPartida, en base a las relaciones ya descritas. El cliente crea su thread ClientController. En la constructora de ClientController se realiza la conexión al servidor mediante la creación del Socket.

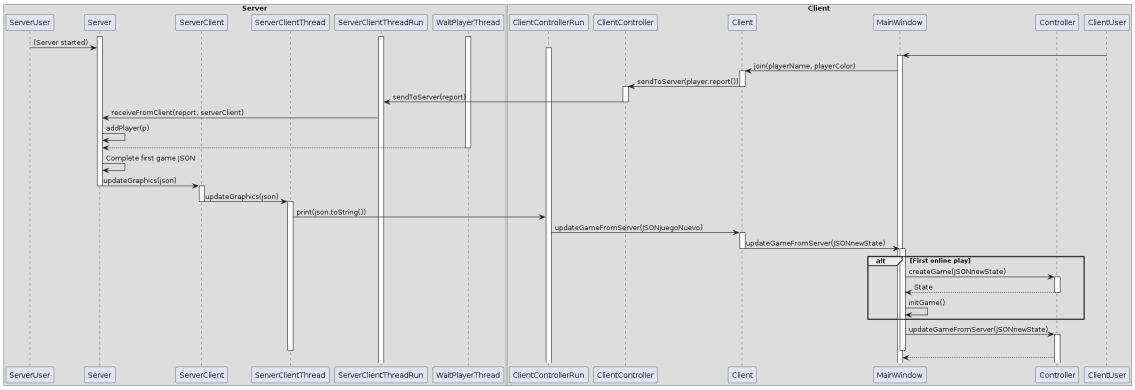
En cuanto esta se realiza con éxito, ocurren dos procesos de forma paralela:

- En el servidor, el método serverSocket.accept() es ejecutado con éxito. Se procede a llamar al método setUpServer del ServerClient específico asociado a esta conexión; se crea después un thread ServerClientThread el cual crea los BufferedReader y PrintWriter oportunos, y ServerClient procede a llamar al run() de este thread (representado en la lifeline ServerClientThreadRun)

- En el cliente, una vez se ha creado el Socket con éxito en la constructora de ClientController, se crean los BufferedReader y PrintWriter oportunos. Se termina la constructora, y acto seguido Client llama al start de ClientController para que empiece a recibir mensajes en su método run().

Si la conexión no se ha realizado bien, se generan las excepciones oportunas que cierran las ventanas de tanto cliente como servidor.

El thread WaitPlayerThread procede a esperar un segundo antes de cerrarse. En este segundo, se espera que el cliente envíe al servidor la información relativa al Player (qué color y qué nombre ha escogido). Esta información pasa a registrarse en la lista de incomingPlayers de forma síncrona mientras WaitPlayerThread espera. El mecanismo pormenorizado es el siguiente:




1. Tras ejecutarse el hilo ClientController, el método empezarPartida de Client se finaliza. MainWindow procede a ejecutar el método join de Client, pasándole como argumentos el nombre y color escogidos.

2. Client pide al thread ClientController que envíe al servidor esta información.

3. ClientController envía la información por medio del método sendToServer

4. El ServerClientThread específico asociado al cliente que envía esta información recibe en su método run el mensaje. Procede a llamar al método asíncrono receiveFromClient de Server.

5. receiveFromClient distingue, observando el JSONObject enviado, si se trata de información de actualización de juego o información respecto a un nuevo jugador añadido. En el caso que nos ocupa, es esta segunda opción; receiveFromClient procede a crear una instancia de Player y añadirla a incomingPlayers (ArrayList de Player).

Todo este proceso tarda (bastante) menos del segundo que espera WaitPlayerThread; se deja un segundo de cortesía para dar un amplio margen a los equipos más lentos.

El método run() de WaitPlayerThread se finaliza y se ejecuta desde waitForPlayers (Server) el método join para que concluya el thread.

Dado la información obtenida de Socket, ServerClient y Player (este último con incomingPlayers, cuyo último elemento añadido es el Player correspondiente al cliente), se rellenan las dos listas Player-Socket y ServerClient-Player de Server con una nueva posición.

Todo esto es una única vuelta del bucle de waitForPlayers. Como hemos comentado, se ejecutarían un número de vueltas equivalente al número de jugadores esperados.

Una vez todos los jugadores se han conectados y las dos listas completadas, se sale del bucle y waitForPlayers procede a crear rellenar el incompleto gameConfigJSON que fue pasado con anterioridad por el constructor de Server; ahora es el momento correcto para hacerlo pues ya se conoce la información de todos los players. Se rellenan los campos pertinentes previamente incompletos de gameConfigJSON; posteriormente, se envía este primer estado del juego a todos los clientes por medio del método updateGraphics.

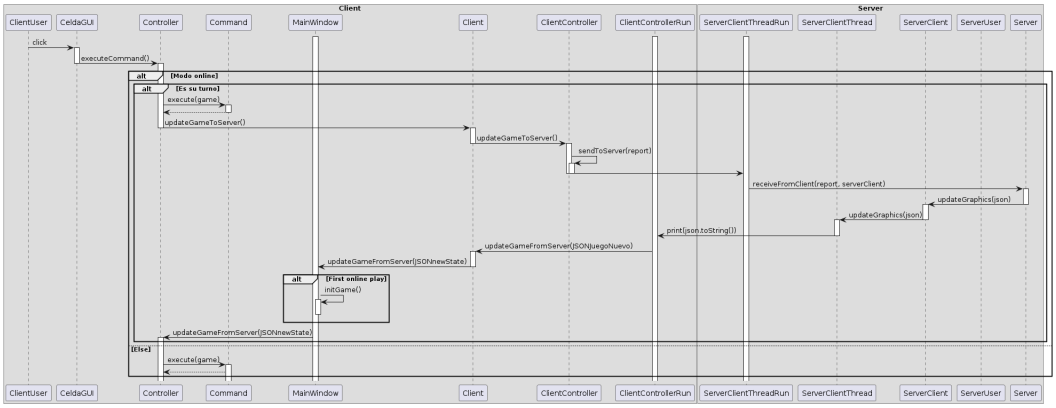
El cliente recibe esta configuración del juego en su thread, se llama al método updateGameFromServer de Client, quien a su vez llama a updateGameFromServer de MainWindow.

En este último método, si es la primera jugada, se llama crea el juego en el controlador, y después se llama a initGame() para inicializar las componentes visuales relativas al juego. En cualquier caso, se llama después al método updateGameFromServer de Controller para actualizar el juego tal y como ordena el JSON pasado como parámetro.

Los mecanismos de este método son crear un nuevo juego a partir del nuevo JSON estado del juego, pasar los observadores del Game antiguo al nuevo, y adjudicar al atributo Game de Controller este nuevo juego.

```
public void updateGameFromServer(JSONObject o) {
  Game newGame = GameBuilder.createGame(o);
  newGame.updateGameFromServer(game.getObserverList());
  game = newGame;
}
```

Falta por especificar cómo se produce una jugada en el modo online y cómo evitar que los clientes jueguen cuando no es su turno.




Se pulsa una celda en el tablero y se pide al controlador que ejecute el comando de poner cubo en la posición pedida, como de costumbre. En caso de que se juegue el modo online, se comprueba si el jugador está legitimado para jugar (es decir, es su turno). Para ello, se comprueba que el jugador turno del juego es el mismo que el jugador del cliente (recordemos que Client tiene el atributo Player del jugador al que corresponde).

De verificarse estas dos condiciones, se ejecuta el comando y se envía el nuevo estado del juego al servidor, quien enviará este nuevo estado al resto de clientes.

De no jugarse el modo online, se ejecuta el comando como de costumbre.

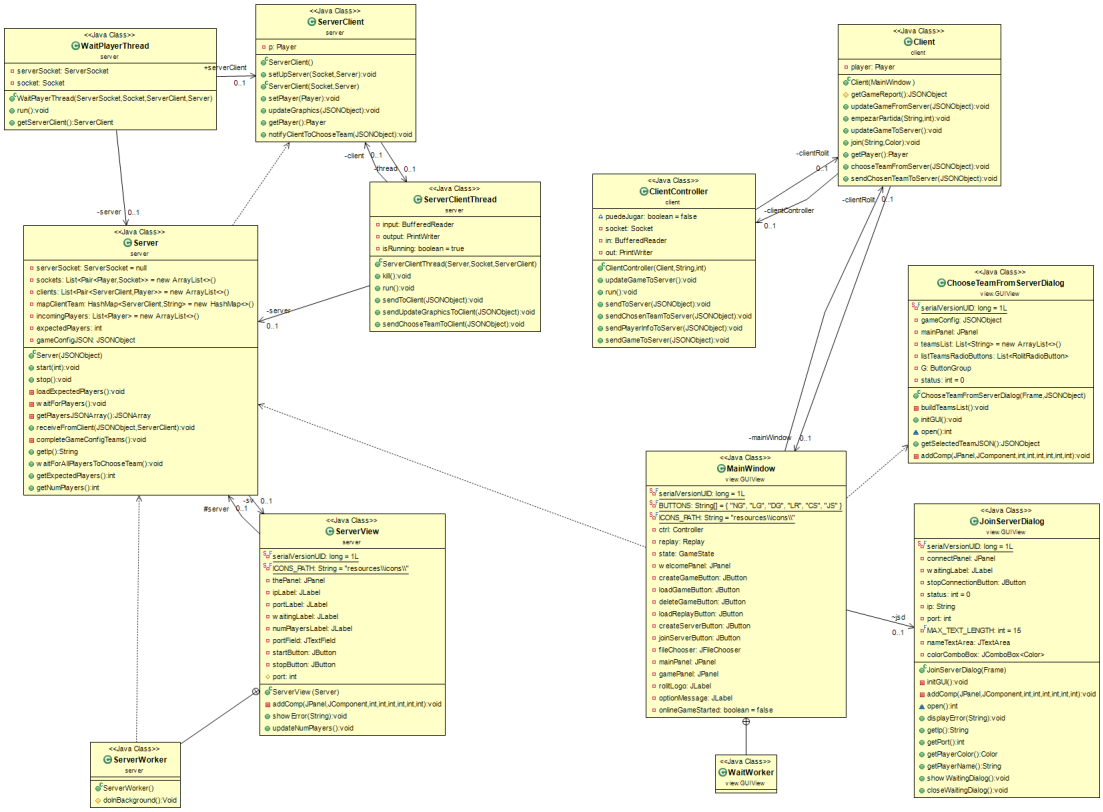
Nótese que este último diagrama de secuencia se representa a Client como entidad abstracta emisora y receptora, realmente un Client sería emisor y el resto de Client serían receptores.

### 6.2.2 Sprint 6

En cuanto a la red, se alcanzan los objetivos propuestos que quedaron pendientes en el Sprint 5. Las características implementadas son:

- Implementación del modo red para GameTeams.
- Creación de diálogos y de código auxiliar para soportar GameTeams.
  - Una clase creada para tal efecto es ChooseTeamFromServerDialog.
- Perfeccionamiento de la estructura y relaciones de las comunicaciones cliente-servidor y viceversa, incluyendo mecanismos como un campo en el JSON mensaje que especifica qué tipo de notificación constituye el mensaje enviado.
- Adaptación de los diálogos GUI de red al nuevo diseño de la interfaz gráfica introducido en este Sprint.

- Se introduce un ServerWorker que soluciona un bug complejo que afectaba a los threads de Red y Swing.
- Simplificación y refactorización del código de Red.




Por último las sucesivas refactorizaciones pugnadas desde otras partes del código (para, por ejemplo soportar la IA) precisan de un debug extensivo en el modo Red. Este debug es satisfactorio.

(INSERTAR DIAGRAMAS DE SECUENCIA UML)

### 6.2.3 Sprint 7

En cuanto a la red, si bien está conclusa llegados a este Sprint, el debug realizado en otras zonas del código influyen directamente en la funcionalidad de Red. Se lleva a cabo un debug rápido pero extensivo para verificar que el modo de juego en red no ha sido afectado, llevándose a cabo con éxito.

Por último, se procede a crear y solventar algunas issues relativas a la experiencia de usuario jugando en red, conllevando una serie de modificaciones que hacen más intuitiva las instrucciones para realizar la conexión.

Los diagramas UML finales son los siguientes: